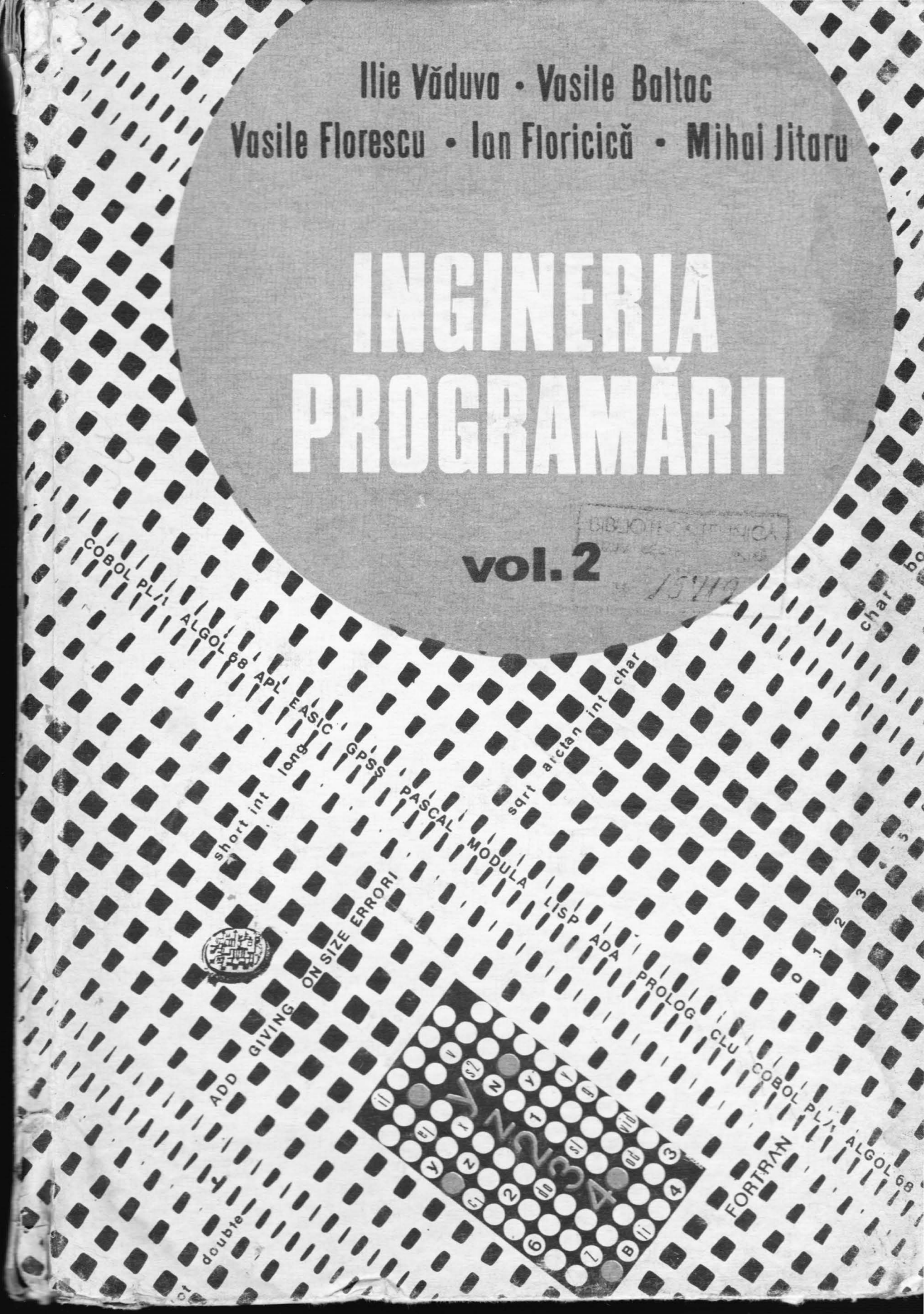
Ilie Văduva • Vasile Baltac
Vasile Florescu • Ion Floricică • Mihai Jitaru

INGINERIA PROGRAMĂRII

vol. 2

BIBLIOTECA
15112



Prefaţă

În primul volum al acestei lucrări au fost prezentate conceptele, principiile, mijloacele, instrumentele, tehnologia cadru şi metodologia de elaborare a programelor începînd cu întocmirea specificaţiilor şi sfîrşind cu codificarea programelor. Capitolul 1 al primului volum a avut rolul de a informa cititorul în legătură cu problematica elaborării programelor, cu cerinţele unor mutaţii în programare care să asigure producerea de programe pe principii industriale, îmbinînd armonios elementele specifice cercetării cu cele specifice producţiei.

Acest de al doilea volum prezintă cititorului în continuare aspecte din problematica elaborării programelor vizînd cu precădere pe cele legate de corectitudinea şi fiabilitatea programelor; testarea şi depanarea programelor; documentarea în programare; implementarea programelor; întreţinerea programelor; calitatea produselor-program; portabilitatea şi adaptabilitatea programelor; evaluarea şi măsurarea performanţelor; conducerea activităţii de elaborare a produselor-program; programarea proceselor concurente, iar în final instrumente şi medii evoluate de programare.

Deseori neglijate în practică, probleme cum ar fi verificarea corectitudinii programelor, documentarea programelor şi automatizarea procesului de elaborare a programelor într-o concepţie unitară (în care intuiţia se îmbină armonios cu experienţa şi asistenţa din partea calculatorului) sînt larg dezbătute. Se fac referiri ample cu privire la crearea cadrului metodologic, organizatoric, instrumentar şi de conducere care să asigure aşezarea pe principii industriale a activităţii de programare, găsind în atelierul integrat de programare cadrul modern de producere a programelor.

Un loc important îl ocupă în lucrare prezentarea celui mai utilizat mediu de programare pe microcalculatoare şi minicalculatoare denumit UNIX. În încheierea volumului se fac scurte referiri la programare folosind limbaje de nivel foarte înalt cu exemple în limbajul dBASE.

Ne exprimăm speranţa că această lucrare va face atractiv subiectul pentru cei interesaţi în construirea programelor informatice de bună calitate, la un cost redus şi în condiţiile folosirii unor medii moderne de programare.

Rămîn totodată deschise studiului problemele descrierii tipurilor şi structurilor de date şi a operaţiilor capabile să permită verificarea complexităţii şi corectitudinii acestora.

Aducem mulţumiri reputatului specialist prof. dr. doc. ing. Mihai Drăgănescu, membru corespondent al Academiei Republicii Socialiste România, care a manifestat interes şi a încurajat cercetarea întreprinsă de autori, referentului ştiinţific dr. ing. Adrian Davidoviciu pentru sugestiile de îmbunătăţire a lucrării şi analiza minuţioasă a manuscrisului, precum şi tuturor celor care prin cercetările lor au contribuit la îmbogăţirea suportului documentar necesar elaborării lucrării.

Se cuvin mulţumiri Editurii Academiei Republicii Socialiste România care a sprijinit cu competenţă apariţia lucrării.

AUTORII

Cuprins

7. **Corectitudinea și fiabilitatea programelor**
   7.1. Introducere 9
   7.2. Metode de verificare a corectitudinii programelor 10
   7.3. Fiabilitatea programelor 26

8. **Testarea și depanarea programelor**
   8.1. Noțiuni de bază 37
   8.2. Strategii de testare 38
   8.3. Moduri de testare 40
   8.4. Metodologia testării structurate 41
   8.5. Depanarea programelor 47
   8.6. Integrarea programelor și testarea de ansamblu 51

9. **Documentarea programelor**
   9.1. Cerințele documentării programelor 53
   9.2. Tipuri de documentații 55
   9.3. Instrumente de documentare 58
   9.4. Autodocumentarea programelor și creșterea accesibilității 59
   9.5. Documentații pe tipuri de utilizatori 61

10. **Implementarea programelor**
   10.1. Considerații generale 62
   10.2. Activități premergătoare implementării 64
   10.3. Activități de implementare 66
   10.4. Aspecte organizatorice 67

11. **Întreținerea programelor**
   11.1. Scopul întreținerii programelor 69
   11.2. Cerințe în întreținerea programelor 70
   11.3. Modificarea programelor 71
   11.4. Instrumente de ajutor în organizarea și întreținerea programelor 73

12. **Calitatea produselor-program**
   12.1. Orientări și realizări în măsurarea calității produselor-program 77
   12.2. Metode, tehnici și instrumente de control a calității 88
   12.3. Un mecanism pentru îmbunătățirea calității programelor .. 90
   12.4. Optimizarea programelor 91

13. **Portabilitatea și adaptabilitatea programelor**
   13.1. Introducere 106
   13.2. Portabilitatea prin limbaje de înalt nivel 107
   13.3. Portabilitatea prin mașini abstracte 109
   13.4. Ierarhia de mașini abstracte 117

13.5. Adaptabilitatea programelor 119
13.6. Portabilitatea sistemului UNIX 121

14. **Evaluarea și măsurarea performanțelor**
14.1. Introducere 122
14.2. Evaluarea performanțelor 123
14.3. Măsurarea performanțelor 137

15. **Conducerea activității de elaborare a produselor-program**
15.1. Introducere 143
15.2. Activități specifice programării 144
15.3. Sistemul de conducere 145
15.4. Organizarea echipei de programare 146
15.5. Indicatori fizici în programare 148
15.6. Factorii care influențează costul programelor 150
15.7. Sistemul de norme și indicatori 153
15.8. Tipizarea și standardizarea — premise ale creșterii eficienței economice în elaborarea produselor-program 157
15.9. Organizarea controlului de calitate 159
15.10. Conducerea prin costuri 163
15.11. Factorii umani în programare 167
15.12. Ergonomia programelor 168
15.13. Dreptul de proprietate în programare și măsuri organizatorice pe linia comercializării programelor 169

16. **Programarea proceselor concurente**
16.1. Introducere 172
16.2. Excluderea reciprocă 174
16.3. Sincronizarea între procese 183
16.4. Comunicarea între procese 192

17. **Instrumente software și medii de programare**
17.1. Introducere 199
17.2. Mediul de programare UNIX 204
17.3. Mediul de programare în limbajul PASCAL 226
17.4. Instrumente software de dezvoltare a programelor pe mini-și microcalculatoare de producție românească 227
17.5. Sisteme integrate de conducere și elaborare a programelor mari: SICEP 232

18. **Programarea folosind limbaje de nivel înalt**
18.1. Introducere 239
18.2. Un exemplu practic de programare folosind un limbaj de nivel înalt 239
18.3. Elemente de bază ale limbajului dBASE 245
18.4. Codificarea structurilor standard de control în limbajul dBASE 247
18.5. Formatele principalelor comenzi dBASE 248
18.6. Exemple practice dezvoltate de proceduri dBASE 256

Anexă 267

Bibliografie 270

Postfață 275

Contents 277

7. Corectitudinea și fiabilitatea programelor

7.1. Introducere

În procesul complex al elaborării programelor, programatorii comit deseori erori. Acestea pot avea printre cauze: slaba pregătire profesională a programatorilor (necunoașterea limbajului, necunoașterea mediilor de programare, necunoașterea metodelor și tehnicilor folosite în programare); neglijență în elaborarea programelor (folosirea unor specificații incomplete, omiterea unor enunțuri în procesul de transformare a specificațiilor de definire a problemei în program calculator) etc. Aceasta face ca programul obținut în urma codificării să nu fie întotdeauna corect.

Se impune deci ca necesară verificarea corectitudinii programelor, care constă în analizarea concordanței între transformările executate de program și specificațiile programului.

Specificațiile programului se exprimă prin relații între variabilele sale care, în scopul formalizării și automatizării procesului de verificare, au o formă matematică.

Există două tipuri de specificații:

— *de intrare:* relații între datele de intrare ale programului;

— *de ieșire:* relațiile privind rezultatul execuției programului.

Aceste specificații sau relații între variabilele programului se mai numesc „aserțiuni". Ele trebuie să caracterizeze programul într-o manieră clară, simplă și completă. De exemplu, să considerăm un program care realizează împărțirea a două numere întregi pozitive x, y și produce cîtul q și restul r. Aserțiunea de intrare este:

$$(x \geqslant 0) \text{ și } (y > 0).$$

Aserțiunea de ieșire este:

$$(x = qy + r) \text{ și } (0 \leqslant r < y).$$

În raport cu noțiunea de aserțiune, un program este corect dacă la terminarea execuției lui aserțiunile sale de ieșire sînt adevărate, în ipoteza că aserțiunile de intrare sînt adevărate.

În general se spune despre un program că este total corect dacă: (a) se termină; (b) rezultatele furnizate sînt cele așteptate de utilizator.

Aplicarea unei metode analitice de verificare a corectitudinii programelor impune deci introducerea unui *model al programului* pe baza căruia condițiile (a) și (b) pot fi formulate riguros. Același model trebuie totodată să pună în evidență caracteristicile generale ale programului.

În practică procedura de demonstrare a corectitudinii programului se separă în două etape:

— corectitudinea parțială: se arată că programul este corect, cu condiția să se termine;

— corectitudinea totală: în plus se arată că programul se termină.

Termenul de *fiabilitate*, spre deosebire de cel de corectitudine, nu are încă o definiție formală și o mare parte din lucrările apărute în acest domeniu nu se pot generaliza. Deși în domeniul fiabilității componentelor hardware s-au înregistrat progrese importante, acestea nu se pot aplica programelor din cauza diferențelor de caracteristici și comportări ce există între piesele hardware și programe. În domeniul hardware, fiabilitatea unei componente se definește ca probabilitatea de funcționare corectă într-o perioadă dată de timp. În general se presupune că piesa hardware este perfectă la început și se deteriorează cu timpul, creind o probabilitate de pană din ce în ce mai mare. Acesta nu este cazul programelor care sînt compuse din instrucțiuni, ce nu îmbătrînesc *). Erorile de programare sînt cauzate de combinarea incorectă a instrucțiunilor, chiar de la început programul fiind pus în funcțiune cu erori. Eliminarea unei erori nu diminuează întotdeauna numărul total al erorilor din program, la corectarea unei erori se pot introduce altele noi. Corectarea programului nu este așa de simplă ca la o piesă hardware, care în cele din urmă poate fi înlocuită.

S-a încercat, așa cum se va vedea mai departe, o abordare formală a fiabilității programelor prin elaborarea unor modele matematice ce definesc relații între parametrii caracteristici fiabilității. De asemenea, fiabilitatea produselor program poate fi privită ca o măsură calitativă a programelor și se poate evalua în acest sens din două puncte de vedere:

a) Numărul erorilor din program (vezi „calitatea programelor"). Trebuie însă știut care este acest număr, ceea ce nu este întotdeauna posibil. Se poate presupune că frecvența de detectare a erorilor în cursul funcționării sale pe o perioadă de timp poate fi utilizată pentru a determina numărul erorilor rămase în program. Metoda nu se poate însă generaliza.

b) Calitatea serviciului adus utilizatorului. Fiabilitatea în acest caz apare ca probabilitatea ca la execuția programului cu anumite date de intrare să se obțină rezultatul așteptat. Ea este dependentă însă de datele de intrare și de mediul în care se execută programul. Această definiție nu se bazează pe numărul de eorori din program: anumite proceduri pot conține un număr ridicat de erori, dar fiabilitatea programului poate fi ridicată, dacă datele de intrare nu activează acele proceduri. De asemenea, fiabilitatea în acest caz nu este o proprietate inerentă programului: acesta poate fi corect, dar funcționarea lui incorectă se poate datora unei erori din sistem, compilator, unitate centrală etc.

7.2. Metode de verificare a corectitudinii programelor

7.2.1. Metoda aserțiunilor invariante

Încă din 1963, Goldstine și von Neumann au arătat că un program poate fi verificat, cel puțin în principiu, dacă se descriu stările tuturor variabilelor din program, după fiecare instrucțiune sau cel puțin, în anumite puncte ale

*). Nu se poate vorbi deci de uzură fizică în programare. Produsele-program nu se defectează și nici nu suferă de uzură fizică.

programului. M. Carthy folosește o metodă similară prin definirea unui vector de stare: fiecare locație din memorie este privită ca o locație dintr-un vector de stare, iar fiecare operație este considerată ca o transformare a vectorului de stare în alt vector de stare. M. Carthy a introdus un formalism pentru definirea programului ca o funcție recursivă, iar problema de verificare a corectitudinii o reduce la o problemă a funcțiilor recursive la care aplică metoda inducției recursive (un set de axiome care transformă o funcție recursivă în alta echivalentă) [33].

Naur [34] generalizează procedeul precedent, considerînd un vector de stare cu valori simbolice, adică a, b, c în loc de —1, 0, 5 etc. Operațiile din program produc expresii simbolice, de tipul $a + b - c$, ca noi elemente ale vectorului de stare, iar în final, se obțin transformările ce au loc asupra variabilelor de intrare.

O simplificare importantă a acestor procedee este adusă de Floyd [35] care dezvoltă așa-zisa *metodă a aserțiunilor*. Ideea este de a urmări numai transformarea a cîtorva variabile (cele semnificative) și a elabora relații numai pentru aceste variabile și numai în anumite puncte esențiale din program. Deci, secvența vectorului de stare se reduce la relațiile între variabilele mai importante ale programului, relații numite aserțiuni. Problema verificării corectitudinii programului se reduce la demonstrarea validității aserțiunilor: dacă fiecare aserțiune este adevărată, atunci programul este corect. Nu există însă nici o procedură de formulare și localizare a aserțiunilor. Ele depind de program și de priceperea și experiența celui care demonstrează. Aceasta se mai numește metoda aserțiunilor invariante pentru că în scopul demonstrării corectitudinii ciclurilor se introduc aserțiuni care se comportă identic ori de cîte ori ajunge controlul la ele, adică sînt invariante. Existența ciclurilor imprimă și un caracter inductiv metodei aserțiunilor, de aceea mai este cunoscută și sub numele de metoda inductivă a aserțiunilor.

Să prezentăm această metodă într-o formă mai sistematică. Un program este un set finit și ordonat de instrucțiuni, cu prima instrucțiune de „start" și ultima instrucțiune de „stop", restul instrucțiunilor fiind de asignare, selecție, salt sau neoperaționale. Prima aserțiune, a_0, poartă numele de aserțiune de intrare și se atașează punctului de start al programului. Ultima aserțiune, a_n, se numește aserțiune de ieșire și se atașează instrucțiunilor de stop. Aserțiunea de intrare specifică domeniul variabilelor de intrare și relațiile dintre ele; aserțiunea de ieșire exprimă rezultatul dorit. Fie secvența $(S_1, S_2, ..., S_n)$ de instrucțiuni din program, cu aserțiunea a_1 atașată instrucțiunii S_1 și aserțiunea a_n atașată instrucțiunii S_n. Se spune că secvența este corectă (sau verificată) dacă presupunînd că a_1 este adevărată, se arată că a_n este adevărată, după execuția secvenței $(S_1, S_2, ..., S_n)$. Demonstrarea corectitudinii programului constă în alegerea și atașarea aserțiunilor de intrare, intermediare și de ieșire în diferite puncte ale programului și verificarea tuturor secvențelor din program prin construirea de condiții de verificare.

Metoda aserțiunilor se aplică astfel:

a) Se alege un subset I de instrucțiuni din program pentru care se atașează aserțiuni.

b) Se atașează aserțiuni pentru fiecare instrucțiune din subsetul I. Definirea aserțiunilor este partea cea mai dificilă a procesului de demonstrare a corectitudinii programului cu atît mai mult cu cît nu poate fi automatizată. Ea depinde de structura programului, de intuiția și experiența progra-

matorului, așa cum se întîmplă la alegerea ipotezelor în inducția matematică. Definirea aserțiunilor depinde, de asemenea, de setul de instrucțiuni I.

c) Se construiesc condițiile de verificare pentru fiecare secvență de instrucțiuni ce pornește de la o aserțiune a_k la altă aserțiune a_{k+1} (fără alte aserțiuni între ele). O condiție de verificare este de forma:

$$a_k \text{ și (asignări) și (condiții de selecție)} \Rightarrow a_{k+1},$$

unde asignările sînt instrucțiunile de atribuire din cadrul secvenței, iar condițiile logice aparțin instrucțiunilor de selecție ce definesc secvența respectivă. Construirea condițiilor de verificare poate fi în întregime automatizată.

d) Se demonstrează corectitudinea parțială a programului prin demonstrarea validității condițiilor de verificare.

e) Se demonstrează corectitudinea totală a programului dacă se arată că programul se termină.

Exemplul 1

Fie programul următor care calculează cîtul Q și restul R ale împărțirii a două numere întregi pozitive X, Y prin metoda scăderilor repetate. Aserțiunile se notează cu a_i, $i = 0, 1, 2$

a_0: $x > 0$ și $y > 0$

1. $i := x$, $q := 0$
2. **cît timp** $i \geqslant y$ **execută**

   a_1 : $x = qy + i$ și $i \geqslant 0$
3. $i := i - y$, $q := q + 1$
4. **sfîrșit ciclu**
5. $r := i$

a_2: $x = qy + r$ și $0 \leqslant r < y$.

Presupunem aserțiunea de intrare a_0 adevărată. Să vedem dacă aserțiunea a_1 este adevărată. Pentru prima dată cînd se ajunge la ea avem condiția de verificare:

$$(x > 0 \text{ și } y > 0) \text{ și } (i := x \text{ și } q := 0) \text{ și } (i \geqslant y) \Rightarrow (x = qy + i \text{ și } i \geqslant 0).$$

În expresia $x = qy + i$ avem: $q = 0$ și $i = x$ (> 0) deci implicația este adevărată. Deoarece bucla se poate executa de un număr de ori trebuie să arătăm că a_1 rămîne adevărată și după ce se reia ciclul. Condiția de verificare este:

$$(x = qy + i \text{ și } i \geqslant 0) \text{ și } (i := i - y \text{ și } q := q + 1) \Rightarrow (x = qy + i \text{ și } i \geqslant 0).$$

În $x = qy + i$ avem $i := i - y$ și $q := q + 1$, deci $x = (q + 1)\,y + i - y = qy + i$, iar i este deja $\geqslant 0$. Deci condiția este verificată.

Aceasta înseamnă că aserțiunea a_1 este întotdeauna adevărată. În continuare trebuie să arătăm că și aserțiunea a_2 este adevărată. Aserțiunea a_2 este atașată instrucțiunii 5, ce poate face parte din două secvențe de instruc-

ţiuni: (S_1, S_5) sau (S_1, S_2, S_3, S_4, S_5) (unde S_3 se poate repeta de mai multe ori), după cum i este mai mare decît y.

Secvenţa (S_1, S_5) există numai dacă $i < y$. Condiţia de verificare este:

$$(x > 0 \text{ şi } y > 0) \text{ şi } (i := x \text{ şi } q := 0) \text{ şi } (i < y) \text{ şi } (r := i) \Rightarrow (x = qy + r \text{ şi } 0 \leqslant r < y),$$

care este evident adevărată.

În secvenţa (S_1, S_2, S_3, S_4, S_5) s-a arătat că a_1 este adevărată. Rămîne de demonstrat că aserţiunea a_1 fiind adevărată, împreună cu condiţia de ieşire din buclă ($i < y$) implică şi aserţiunea a_2 ca fiind adevărată. Condiţia de verificare este:

$$(x = qy + i \text{ şi } i \geqslant 0) \text{ şi } (i < y) \text{ şi } (r := i) \Rightarrow (x = qy + r \text{ şi } 0 \leqslant r < y)$$

Ea este evident adevărată.

În felul acesta s-a demonstrat corectitudinea parţială a programului. Pentru a avea demonstrarea corectitudinii totale trebuie arătat că programul se termină. Se observă că bucla 2, 3, 4 se execută atît cît $i \geqslant y$; dar la fiecare execuţie a buclei, i scade cu y ($y > 0$), iar y rămîne constant, deci în mod obligatoriu se ajunge ca $i < y$, ceea ce asigură ieşirea din buclă, deci programul se termină după un număr finit de iteraţii. Aceasta completează demonstrarea corectitudinii totale.

Exemplul 2

Fie programul următor, care calculează cel mai mare divizor comun (c.m.m.d.c.) a două numere întregi pozitive x, y. Aserţiunile se notează cu a_i, $i = 0, 1, 2$.

```
a0 : x ≥ 0 şi y ≥ 0 şi (x ≠ 0 sau y ≠ 0)
     u: = x
     v: = y
teste:
a1: u ≥ 0 şi v ≥ 0 şi (u ≠ 0 sau v ≠ 0) şi
max (t: t | u şi t | v) = max (t: t | x şi t | y)
dacă u = 0 atunci salt la sfîrşit
dacă v ≥ u atunci v: = v − u altfel (u, v): = (v, u)
salt la teste
sfîrşit :
a2: v = max (t: t | x şi t | y)
imprimă v
```

Prin notaţia max (t:t | u şi t | v) se indică valoarea maximă a lui t care împarte exact şi pe u şi pe v, iar prin (u, v) : = (v, u) se indică schimbarea de valori între u şi v.

Presupunem adevărată aserţiunea de intrare. Să arătăm că aserţiunea intermediară a_1 este adevărată. Pentru prima dată cînd se ajunge la ea avem condiţia de verificare următoare:

$$(x \geqslant 0 \text{ şi } y \geqslant 0) \text{ şi } (x \neq 0 \text{ sau } y \neq 0) \text{ şi } (u := x \text{ şi } v := y) \Rightarrow$$
$$(u \geqslant 0 \text{ şi } v \geqslant 0) \text{ şi } (u \neq 0 \text{ sau } v \neq 0) \text{ şi } \max(t:t \mid u \text{ şi } t \mid v) = \max(t:t \mid x \text{ şi } t \mid y).$$

Ea este evident adevărată (u fiind egal cu x, iar v fiind egal cu y).

În continuare trebuie să arătăm că aserţiunea a_1 rămîne adevărată de fiecare dată cînd se revine în buclă la eticheta „teste".

În cadrul buclei distingem două cazuri: $v \geqslant u$ și $v < u$

Cazul $v \geqslant u$

Se execută instrucțiunea v: = v — u, deci condiția de verificare este:

$$(u \geqslant 0 \text{ și } v \geqslant 0) \text{ și } (u \neq 0 \text{ sau } v \neq 0) \text{ și } \max(t{:}t \mid u \text{ și } t \mid v) = \max(t{:}t \mid x \text{ și } t \mid y) \text{ și } (u \neq 0 \text{ și } v \geqslant u) \Rightarrow (u \geqslant 0 \text{ și } v - u \geqslant 0) \text{ și } (u \neq 0 \text{ sau } v - u \neq 0) \text{ și } \max(t{:}t \mid u \text{ și } t \mid v - u) = \max(t{:}t \mid x \text{ și } t \mid y).$$

În relația implicată s-a înlocuit v cu $v—u$, conform instrucțiunii de atribuire v:=v—u. Prin aplicarea teoremei divizorului (dacă t împarte pe u și v, atunci împarte și diferența lor) rezultă validitatea condiției de verificare.

Cazul $v < u$

Ținînd cont că se schimbă u cu v, condiția de verificare este:

$$(u \geqslant v \text{ și } v \geqslant 0) \text{ și } (u \neq 0 \text{ sau } v \neq 0) \text{ și } \max(t{:}t \mid u \text{ și } t \mid v) = \max(t{:}t \mid x \text{ și } t \mid y) \text{ și } (u \neq 0 \text{ și } v < u) \Rightarrow (v \neq 0 \text{ și } u \neq 0) \text{ și } (v \neq 0 \text{ sau } u \neq 0) \text{ și } \max(t{:}t \mid v \text{ și } t \mid u) = \max(t{:}t \mid x \text{ și } t \mid y).$$

Ea este evident adevărată.

Pentru a termina demonstrarea corectitudinii parțiale a programului trebuie arătat că dacă aserțiunea intermediară a_1 este adevărată, atunci și aserțiunea de ieșire a_2 este adevărată. Condiția de verificare este:

$$(u \geqslant 0 \text{ și } v \geqslant 0) \text{ și } (u \neq 0 \text{ sau } v \neq 0) \text{ și } \max(t{:}t \mid u \text{ și } t \mid v) = \max(t{:}t \mid x \text{ și } t \mid y) \text{ și } (u = 0) \Rightarrow v = \max(t{:}t \mid x \text{ și } t \mid y)$$

care este adevărată, pentru că:

- u fiind nul rezultă cu $v > 0$ (din relațiile $v \geqslant 0$ și ($u \neq 0$ sau $v \neq 0$));
- orice întreg împarte pe zero, deci relația

max (t:t | u și t | v) = max (t:t | x și t | y) devine:

max (t:t | v) = max(t:t | x și t | y);

- v fiind > 0 atunci max(t:t | v) = v (un întreg pozitiv este propriu său c.m.m.d.c.). Deci v = max(t:t | x și t | y).

Notăm rolul important al aserțiunii intermediare a_1; demonstrarea corectitudinii secvenței de instrucțiuni din buclă s-a făcut în numai două cazuri:

— de la aserțiunea a_1 la ea însăși, prin condiția $v \geqslant u$;

— de la aserțiunea a_1 la ea însăși prin condiția $v < u$.

Dacă nu s-ar fi introdus această aserțiune ar fi trebuit luate în considerare infinitele posibilități de trecere de la aserțiunea de intrare a_0 la cea de ieșire a_2, corespunzător tuturor datelor pentru care se poate executa programul.

În continuare, pentru a demonstra că programul este în totalitate corect trebuie să arătăm că se termină. Am presupus că x și y sînt întregi pozitivi. Se observă că de fiecare dată cînd se trece prin buclă se reduce sau u sau v: dacă $v \geqslant u$ atunci se reduce v ($v := v - u$), iar dacă $v < u$ se reduce u (prin schimbarea valorilor între u și v). Argumentul ieșirii din buclă îl constituie faptul că u și v nu pot scade la infinit, fără a deveni negativi, ceea ce ar contrazice aserțiunea intermediară a_1, care am arătat că este adevărată.

7.2.2. Metoda inducţiei convenabile

Principiul inducţiei convenabile (well-founded induction) permite demonstrarea corectitudinii parţiale a programelor printr-o metodă asemănătoare inducţiei complete. Aici vom face numai o trecere în revistă a acestei metode deoarece procesul de demonstrare este, în general, complex prin faptul că tratează programul ca o entitate, fără o descompunere în mai multe etape.

Mai întîi să definim mulţimile şi ordinile convenabile [37]. Se ştie că o mulţime M prevăzută cu o relaţie binară „$<$" este parţial ordonată dacă relaţia este:

a) tranzitivă: pentru orice $x, y, z \in M$, dacă $x < y$ şi $y < z$ atunci $x < z$;

b) antisimetrică: pentru orice $x, y \in M$, dacă $x < y$ atunci $y \nless x$;

c) nonreflexivă: pentru orice $x \in M$, $x \nless x$.

O mulţime parţial ordonată ce nu conţine nici un şir descrescător infinit se numeşte mulţime convenabil ordonată, iar relaţia de ordine se numeşte ordin convenabil.

De exemplu, mulţimea numerelor întregi pozitive prevăzută cu relaţia de ordine cunoscută $<$ este convenabil ordonată.

Mulţimea $N \times N$, a perechilor de numere naturale, prevăzută cu relaţia de ordine lexicografică:

$$(x_1, y_1) < (x_2, y_2) \Leftrightarrow x_1 < x_2 \text{ sau } (x_1 = x_2 \text{ şi } y_1 < y_2)$$

este, de asemenea, o mulţime convenabilă, iar relaţia de ordine lexicografică este un ordin convenabil. Mulţimile convenabile au fost utilizate pentru prima dată de Floyd în scopul demonstrării terminării execuţiei programelor [35].

Principiul inducţiei convenabile se enunţă astfel: fie M o mulţime prevăzută cu un ordin convenabil $<$; pentru a demonstra că proprietatea p este adevărată pentru orice element din M se consideră un element arbitrar $x \in M$ şi se demonstrează că $p(x)$ este adevărată în ipoteza că $p(y)$ este adevărată pentru orice element $y \in M$ mai „mic" decît x ($y < x$). Cu alte cuvinte, pentru a arăta că orice element dintr-o mulţime convenabilă posedă o proprietate se alege un element arbitrar x din mulţime, se presupune că orice element mai „mic" ca x satisface proprietatea şi se demonstrează că x satisface şi el proprietatea.

De exemplu, fie programul de calcul al celui mai mare divizor comun a două numere întregi pozitive, scris într-o nouă formă faţă de cea prezentată anterior. Aserţiunea de intrare este a_0, iar cea de ieşire a_1

$a_0 : x_0 > 0$ şi $y_0 > 0$

$x := x_0;\ y := y_0$

teste: **dacă** x = y **atunci salt la** sfîrşit;

dacă x > y **atunci** x: = x − y **altfel** y: = y − x;

salt la teste;

sfîrşit:

a_1 : c.m.m.d.c. (x, y) = c.m.m.d.c. (x_0, y_0)

Pentru demonstrarea corectitudinii programului considerăm mulţimea $N \times N$ prevăzută cu relaţia de ordine lexicografică, care este o relaţie convenabilă. Vrem să arătăm că programul este corect pentru o valoare arbitrară $(x, y) \in N \times N$, adică presupunînd aserţiunea de intrare adevărată rezultă că şi aserţiunea de ieşire este adevărată.

Conform principiului inducției convenabile presupunem că programul este corect pentru orice $(x', y') < (x, y)$, care satisface aserțiunea de intrare.

Dacă $x_0 = y_0$ atunci demonstrația este terminată pentru că c.m.m.d.c. $(x_0, x_0) = x_0$.

Dacă $x_0 \neq y_0$ atunci avem:

— fie $x > y$, deci $x' = x - y$ și $y' = y$,

— fie $x < y$, deci $x' = x$ și $y' = y - x$.

În oricare din cele două cazuri, avem adevărată inegalitatea:

$$(0, 0) < (x', y') < (x, y).$$

Conform ipotezei, programul este corect pentru $(x', y') \in N \times N$, deci:

$$\text{c.m.m.d.c. } (x', y') = \text{c.m.m.d.c. } (x_0, y_0).$$

Dar conform teoremei divizorului avem:

$$\text{c.m.m.d.c. } (x', y') = \text{c.m.m.d.c. } (x, y).$$

Deci, în final:

$$\text{c.m.m.d.c. } (x, y) = \text{c.m.m.d.c. } (x_0, y_0),$$

adică programul este corect pentru (x, y) arbitrar ales, ceea ce înseamnă că este corect pentru orice $(x, y) \in N \times N$.

Deoarece la eticheta „sfîrșit" avem $x = y$, rezultă că c.m.m.d.c. a valorilor x_0, y_0 va fi x:

$$x = \text{c.m.m.d.c. } (x, y) = \text{c.m.m.d.c. } (x_0, y_0).$$

O aplicație interesantă a mulțimilor convenabile o constituie formularea unei metode riguroase de demonstrare a terminării unui program cu cicluri. Se procedează astfel:

a) se definește o mulțime convenabilă cu un ordin convenabil;

b) se definește o expresie de terminare $e(x)$ astfel încît la fiecare reluare a ciclului valoarea expresiei să aparțină mulțimii convenabile, dar să scadă în raport cu relația $<$. Aceasta stabilește terminarea programului pentru că dacă nu s-ar ieși din buclă ar exista o secvență descrescătoare infinită în mulțimea convenabilă, ceea ce ar contrazice definiția ei.

De exemplu, în cazul programului precedent pentru care s-a considerat mulțimea convenabilă $N \times N$ cu relația de ordine lexicografică se consideră ca expresie de terminare: $e(x, y) = (x, y)$.

Se observă că:

— dacă $x > y$ atunci $(x - y, y) \in N \times N$ și $(x - y, y) < (x, y)$,

— dacă $x < y$ atunci $(x, y - x) \in N \times N$ și $(x, y - x) < (x, y)$.

Deci programul se termină după un număr finit de iterații.

Terminarea programului se poate demonstra și prin metoda inducției convenabile. De exemplu, pentru același program ca mai sus se consideră mulțimea convenabilă $N \times N$ cu relația de ordine convenabilă următoare [36]:

$$(x', y') \ll (x, y) \Leftrightarrow e(x', y') < e(x, y),$$

unde $(x', y'), (x, y) \in N \times N$, iar e este o expresie de terminare și $<$ este o relație de ordine convenabilă a mulțimii $N \times N$. Fie $e(x, y) = (x, y)$ și relația de ordine lexicografică. Considerăm un element arbitrar $(x, y) \in N \times N$ și presupunem că programul se termină pentru orice (x', y') mai „mic" ca (x, y)

$((x', y') \ll (x, y))$. Ca și mai sus se arată că la o trecere în buclă (x, y) se transformă în (x', y'), astfel încît:

$$e(x', y') < e(x, y)$$
$$(x', y') \ll (x, y),$$

ceea ce înseamnă, conform ipotezei, că programul se termină și pentru (x, y).

7.2.3. Metoda aserțiunilor intermitente

Pînă acum am descompus procesul de demonstrare a corectitudinii programului în două etape: programul produce rezultatul dorit (dacă se termină); programul se termină.

Burstall [38] introduce o nouă metodă dezvoltată ulterior de Manna [36] prin care stabilește corectitudinea totală a programului în cadrul unei singure demonstrații. Pentru aceasta se introduce noțiunea de aserțiune intermitentă, care este verificată cel puțin o dată atunci cînd execuția programului ajunge la instrucțiunea căreia i-a fost atașată. Aserțiunea de ieșire devine aserțiune intermitentă ce ia valoarea „adevărat" chiar în momentul în care programul se termină. Se demonstrează simultan corectitudinea parțială a programului și terminarea lui, stabilindu-se astfel corectitudinea totală.

Se introduce notația: „uneori a_i la l", pentru a arăta că uneori aserțiunea a_i este adevărată atunci cînd execuția programului ajunge la eticheta l. Tot astfel se poate folosi notația: „totdeauna a_i la l", pentru a indica invarianța aserțiunii a_i la eticheta l.

Notînd cu a_0 aserțiunea de intrare și a_n aserțiunea de ieșire atașate respectiv punctelor de start și stop ale programului, atunci corectitudinea programului se poate exprima astfel:

dacă uneori a_0 la „start"
atunci uneori a_n la „stop".

Considerăm programul de calcul al celui mai mare divizor comun a două numere întregi pozitive x_0, y_0 și prezentăm demonstrația corectitudinii lui prin metoda aserțiunilor intermitente, așa cum apare tratată de Manna și Waldinger [36]:

```
a0 :    x0 > 0 și y0 > 0
start:  x: = x0; y: = y0
rep:    dacă x = y atunci salt la stop;
etx:    dacă x > y atunci (x: = x − y; salt la rep);
ety:    dacă x < y atunci (y: = y − x; salt la rep);
stop:
a1:     x = c.m.m.d.c. (x0, y0)
tipărește x
```

Corectitudinea programului se poate exprima astfel:

dacă uneori $x_0 > 0$ și $y_0 > 0$ la start
atunci uneori x = c.m.m.d.c. (x_0, y_0) la stop.

Această afirmație stabilește corectitudinea parțială a programului, dar și terminarea lui pentru că se precizează că se poate ajunge la eticheta stop care este punctul de ieșire din program. Pentru a demonstra aceasta, ne folosim de următoarea lemă:

dacă uneori $x = a$ și $y = b$ și $a, b > 0$ la eticheta rep
sau uneori $x = a$ și $y = b$ și $a, b > 0$ la eticheta etx

sau uneori $x = a$ și $y = b$ și $a, b > 0$ la eticheta ety
atunci uneori $x =$ c.m.m.d.c. (a, b) la eticheta stop.

Este suficient de a demonstra această lemă pentru a demonstra corectitudinea programului, deoarece la eticheta start avem $x = x_0$, $y = y_0$, $x_0 > 0$, $y_0 > 0$, iar controlul ajunge la eticheta rep astfel încît ipoteza:

uneori $x = x_0$ și $y = y_0$ și $x_0, y_0 > 0$ la eticheta rep este îndeplinită, ceea ce înseamnă că:

uneori $x =$ c.m.m.d.c. (x_0, y_0) la eticheta stop.

Pentru a demonstra lema se folosește metoda inducției convenabile pentru mulțimea N × N, prevăzută cu relația de ordine convenabilă definită astfel:

$$(x', y') < (x, y) \text{ dacă } x' + y' < x + y.$$

Fie $(x = a, y = b)$ și presupunem că lema este adevărată pentru orice $(x' = a', y' = b')$ mai „mic" ca (x, y), adică:

$$a' + b' < a + b.$$

Se disting 3 cazuri:

1) $a = b$.

Indiferent dacă controlul este la rep, etx sau ety se ajunge la eticheta stop cu $x = a = b$, deci:

uneori $x = a = b$ la eticheta stop.

În acest caz $x = a =$ c.m.m.d.c. $(a, a) =$ c.m.m.d.c. (a, b), deci

uneori $x =$ c.m.m.d.c. (a, b) la eticheta stop

care este de fapt concluzia lemei.

2) $a > b$.

Indiferent dacă controlul este la rep, etx sau ety se ajunge la eticheta etx unde x se reduce de la a la $a—b$, deci:

uneori $x = a - b$ și $y = b$ la eticheta etx.

Notăm $a' = a - b$ și $b' = b$. Deci:

$$a', b' > 0$$
$$a' + b' < a + b$$

c.m.m.d.c. $(a'b') =$ c.m.m.d.c. $(a - b, b) =$ c.m.m.d.c. (a, b).

Deoarece $a', b' > 0$ și $a' + b' < a + b$ din ipoteza de inducție rezultă că:

uneori $x =$ c.m.m.d.c. (a', b') la eticheta stop

Conform egalității de mai sus rezultă:

uneori $x =$ c.m.m.d.c. (a, b) la eticheta stop

care este de fapt concluzia lemei.

3) $a < b$.

Se rezolvă prin simetrie ca și cazul precedent. Deci lema este adevărată, adică programul este corect.

Manna notează că metoda aserțiunilor intermitente este mai puternică decît metoda aserțiunilor invariante, deoarece demonstrațiile de corectitudine făcute prin metoda aserțiunilor invariante pot fi făcute și prin metoda aserțiunilor intermitente, dar reciproca nu este întotdeauna adevărată.

*

Toate exemplele de programe prezentate în paragrafele precedente au fost extrem de simple, de cîteva instrucțiuni, dar demonstrațiile au fost, uneori, destul de elaborate. Aceasta scoate în evidență gradul înalt de complexitate al metodelor de verificare a corectitudinii programelor. Practic este imposibil să se demonstreze corectitudinea unui sistem complex de programe.

Au fost construite mai multe sisteme software pentru demonstrarea semi-automată a corectitudinii programelor [39, 40]. Condițiile de verificare sînt generate automat, dar aserțiunile sînt furnizate de programator. Programatorul mai poate furniza axiome sau teoreme pe baza cărora calculatorul poate furniza demonstrația completă a programului. Procesul stabilirii aserțiunilor este la fel de greoi și complicat ca și cel al elaborării programului. Se pare că singura cale de avansare în acest domeniu ar fi automatizarea completă; cere însă mai mult formalism în demonstrații.

Dar, pe de altă parte, în calea procesului de demonstrare a corectitudinii programelor există bariere ce nu pot fi înlăturate practic de nici o inovație tehnică. De exemplu:

a) Nu putem fi siguri că specificațiile programului sînt corecte. A verifica un program înseamnă a arăta că aceasta satisface specificațiile sale. Nu se poate ști însă că aceste specificații reflectă întocmai intențiile conceptorului. Dacă s-a făcut o eroare în definirea lor, aceasta nu poate fi detectată de sistemul automat de verificare.

b) Nici un sistem de verificare nu poate afirma că un program este corect, sau nu se poate construi un program care să certifice că orice alt program este corect sau nu. Pentru orice sistem de verificare se poate găsi un program corect despre care sistemul nu poate decide dacă este corect sau nu.

c) Nu putem fi siguri că sistemul de verificare este corect.Acesta este un complex de programe construite de programatori cu toate neajunsurile inerente legate de activitatea de programare, deci inclusiv introducerea de erori.

Mulți autori nu sînt de acord cu sistemele de verificare automată și recomandă metoda testării ca mijloc de creștere a fiabilității programelor. Se știe însă că testarea nu certifică exactitudinea programelor: ea arată prezența erorilor dar nu absența lor.

7.2.4. Execuția simbolică

Testarea programelor și demonstrarea corectitudinii pot fi considerate ca două extreme pentru procesul de verificare al programelor. Metoda execuției simbolice se situează între aceste două procedee. Tehnica testării presupune analiza rezultatelor obținute pe baza execuției programului cu date concrete de intrare. Demonstrarea corectitudinii arată că programul se execută corect pentru un set larg de date de intrare. Execuția simbolică „analizează" programul pentru un grup de date de intrare și, în principiu, ea este echivalentă cu un număr foarte mare de testări cu date concrete.

Execuția simbolică este o extensie naturală a execuției normale cu deosebirea că operatorii acceptă variabile de intrare simbolice și produc la ieșire formule simbolice. De exemplu, dacă de fiecare dată se cere o nouă valoare de intrare pentru program aceasta se furnizează din lista de simboli $a_1, a_2, a_3, \ldots$.

Instrucțiunea GØTØ funcționează exact ca și în cazul execuției normale, transferînd controlul la instrucțiunea etichetată. Evaluarea expresiilor se face cu valorile simbolice ale variabilelor, adică $a_1 + a_2 - a_3$ în loc de $1 + 2 - 5$, de exemplu.

În continuare vom prezenta principiul execuției simbolice [41] rezumîndu-ne la un limbaj simplu ce posedă: variabile de tip întreg; instrucțiuni de selecție de tip „if then else"; instrucțiuni de salt „goto"; etichete pentru

instrucțiuni; operații de citire, scriere, apel de proceduri; operații aritmetice (+, —, x, /); expresii booleene pentru testarea unei valori aritmetice în raport cu 0.

Vectorul de stare al programului cuprinde:

- valorile simbolice ale variabilelor din program;
- contorul de instrucțiuni ce conține adresa următoarei instrucțiuni de executat;
- o variabilă (expresie) booleană, inițializată la „adevărat", pentru selectarea diverselor ramuri din program generate de instrucțiuni de tip IF. Ea cumulează condițiile pe care trebuie să le satisfacă variabilele de intrare pentru ca execuția programului să urmeze o anumită cale. Vom nota această variabilă cu ps (predicat de selecție).

Execuția simbolică a instrucțiunilor de tip IF începe, ca și în cazul execuției normale, prin evaluarea expresiei sale booleene prin înlocuirea variabilelor cu valorile lor simbolice. Dacă variabilele iau valori din lista $a_1, a_2, a_3, \ldots$ atunci evaluarea expresiei booleene produce un polinom P în a_i, iar condiția finală va fi de forma $P \leqslant 0$ sau $P > 0$. Notăm o astfel de expresie cu c. Utilizînd variabila ps de selecție a unei căi se formează expresiile:

a) $ps \Rightarrow c$

b) $ps \Rightarrow \neg c$.

Cel puțin una din aceste expresii va fi adevărată (se exclude cazul în care ps este fals). Atunci cînd doar o expresie este adevărată se continuă execuția pe ramura corespunzătoare. În acest caz se spune că instrucțiunea IF este rezolvabilă. Atunci cînd nici una din expresiile de mai sus nu este adevărată, ceea ce înseamnă că pentru anumite date concrete se urmează alternativa a), iar pentru altele se urmează b) va trebui continuată execuția simbolică simultan pe cele două ramuri ale instrucțiunii IF. În acest caz se spune că instrucțiunea IF este nerezolvabilă (sau paralelă). La alegerea unei ramuri se cumulează noua condiție din instrucțiunea IF în ps prin asignarea:

ps: = ps și c sau ps: = ps și ¬ c.

Execuția simbolică a instrucțiunii DØ WHILE urmează aceleași reguli ca și la instrucțiunea IF.

do while exp. booleeană;
bloc de instrucțiuni
end;

Evaluarea expresiei booleene se face ca la instrucțiunea IF iar execuția blocului de instrucțiuni depinde de rezultatul evaluării.

Exemple.

1. Fie programul de mai jos scris în PL/1 care calculează suma pătratelor a trei numere întregi: A, B, C.

```
1. SUMP: PROCEDURE (A, B, C);
2.    X = A * A;
3.    Y = B * B;
4.    Z = C * C + X + Y;
5.    RETURN (Z);
6. END;
```

Execuția simbolică a programului este redată în tabelul 7.1.

Tabelul 7.1. Execuția simbolică a programului din exemplul 1

Nr. instr.	X	Y	Z	A	B	C	ps
1				a	b	c	adev.
2	a^2			a	b	c	adev.
3	a^2	b^2		a	b	c	adev.
4	a^2	b^2	$a^2 + b^2 + c^2$	a	b	c	adev.
5		rezultat $a^2 + b^2 + c^2$					

2. Programul următor calculează a^b unde a, b, sînt numere întregi pozitive.

```
1. PUTERE: PROCEDURE (A, B);
2.    X = 1;
3.    Y = 1;
4.    WHILE (Y ≤ B) DØ;
5.       X = X * A;
6.       Y = Y + 1;
7.    END;
8.    RETURN (X);
9. END;
```

Execuția simbolică a programului se prezintă în tabelul 7.2.

Tabelul 7.2. Execuția simbolică a programului din exemplul 2

Nr. instr.	A	B	X	Y	ps
1	a	b			adev.
2	a	b	1		adev.
3	a	b	1	1	adev.
4	cazul $\neg(1 \leqslant b)$				
	a	b	1	1	$\neg(1 \leqslant b)$
8	rezultat 1				
4	cazul $1 \leqslant b$				
	a	b	1	1	$1 \leqslant b$
5	a	b	a	1	$1 \leqslant b$
6	a	b	a	2	$1 \leqslant b$
7	a	b	a	2	$1 \leqslant b$
4	cazul $\neg(2 \leqslant b)$				
	a	b	a	2	$1 \leqslant b$ și $\neg(2 \leqslant b)$
8	rezultat a				
4	cazul $2 \leqslant b$				
	a	b	a	2	$1 \leqslant b$ și $2 \leqslant b$
	etc.				

Se observă că execuția simbolică a programului din tabela 7.2 poate continua la infinit.

Arborele de execuție simbolică

Execuția simbolică a programului se poate reprezenta printr-un arbore astfel: fiecare instrucțiune reprezintă un nod etichetat cu numărul instrucțiunii, iar transferul de la o instrucțiune la alta se reprezintă printr-un arc orientat. Pentru fiecare instrucțiune de tip IF nerezolvabilă se asociază două arcuri, unul marcat cu A (adevărat) și celălalt marcat cu F (fals), corespunzătoare

tor blocului THEN, respectiv ELSE. De asemenea, se poate reprezenta la fiecare nod starea curentă a execuției: valorile variabilelor, contorul de instrucțiuni și predicatul de selecție (ps). De exemplu, arborele de execuție simbolică al procedurii PUTERE este cel din figura 7.1.

Demonstrarea corectitudinii programelor prin metoda execuției simbolice se bazează în întregime pe metoda aserțiunilor invariante a lui Floyd [35].

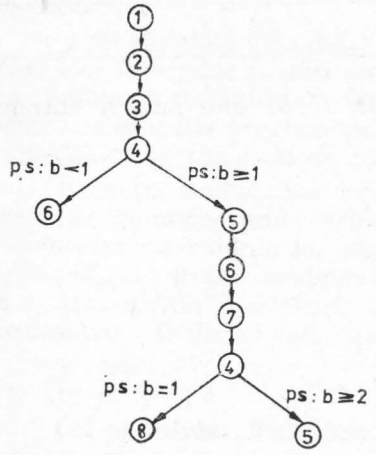


Fig. 7.1. — Arborele de execuție simbolică al procedurii PUTERE.

Execuția simbolică constituie mijlocul prin care se arată dacă justețea aserțiunilor de intrare garantează justețea aserțiunilor de ieșire. Pe de altă parte se generează automat condițiile de verificare.

Dacă arborele de execuție simbolică este finit atunci se poate da o dovadă de corectitudine a programului printr-o simplă parcurgere finită a lui. În caz contrar trebuie utilizate aserțiuni adiționale inductive.

În scopul demonstrării corectitudinii programului prin această metodă se introduc în limbajul de programare trei noi instrucțiuni:

— ASSUME (p): corespunde aserțiunii de intrare a_0,

— ASSERT (q): corespunde unei aserțiuni intermediare a_i,

— PRØVE (r): corespunde aserțiunii de ieșire a_n,

p, q, r sînt expresii booleene (condiții asupra variabilelor).

În execuția unei instrucțiuni ASSUME (p) se evaluează predicatul p utilizînd valorile curente ale variabilelor din program și rezultatul se introduce în predicatul de selecție ps:

$$ps := ps \text{ și } p.$$

Instrucțiunea PRØVE (r) se execută formîndu-se expresia: $ps \Rightarrow r$ și se caută să se demonstreze dacă este adevărată sau falsă.

Pentru demonstrarea corectitudinii programului prin metoda execuției simbolice se procedează astfel:

a) se introduce instrucțiunea ASSUME la începutul programului și instrucțiunea PRØVE la sfîrșitul programului;

b) se introduc instrucțiuni ASSERT în diferite puncte ale programului;

c) se inițializează ps cu valoarea „adevărat" și toate variabilele de intrare cu valori simbolice distincte;

d) se execută simbolic programul; de fiecare dată cînd se întîlnește o instrucțiune IF nerezolvabilă se impune valoarea „adevărat" sau „fals" după cum se dorește să se urmeze o ramură sau alta;

e) se evaluează expresia booleeană implicată de instrucțiunea PRØVE; dacă este adevărată atunci ramura parcursă este corectă.

Procesul de demonstare a corectitudinii programelor prin execuția simbolică este asemănător cu cel al aserțiunilor invariante. Deosebirea constă în faptul că predicatul de selecție și expresiile ce trebuie demonstrate decurg din program, pe cînd la Floyd aserțiunile sînt dependente de problemă. De aici rezultă și interesul execuției simbolice. Exemplu [41]:

Fie programul de calcul al celui mai mare divizor comun a două numere întregi pozitive:

```
1.  DIVIZOR: PRØCEDURE (X, Y);
2.      ASSUME (X > 0) şi Y > 0);
3.      A = X;
4.      B = Y;
5.      WHILE (A ≠ B) DØ;
6.          IF A > B
7.              THEN A = A — B
8.              ELSE B = B — A;
9.      END;
10.     PRØVE (A = CMMDC (X, Y);
11.     RETURN (A);
12. END
```

Vom folosi următoarele notaţii pentru arborele de execuţie simbolică: dacă $a > 0$ şi $b > 0$ vom scrie $a, b > 0$, iar cmmdc (a, b) se va nota $D(a, b)$.

Dacă se construieşte arborele de execuţie simbolică, acesta va fi infinit (fig. 7.2) din cauza valorilor simbolice ale variabilelor de intrare, ceea ce nu înseamnă că şi execuţia cu date concrete va fi infinită. Reciproca este însă adevărată: un program cu execuţie normală infinită va avea şi execuţie simbolică infinită.

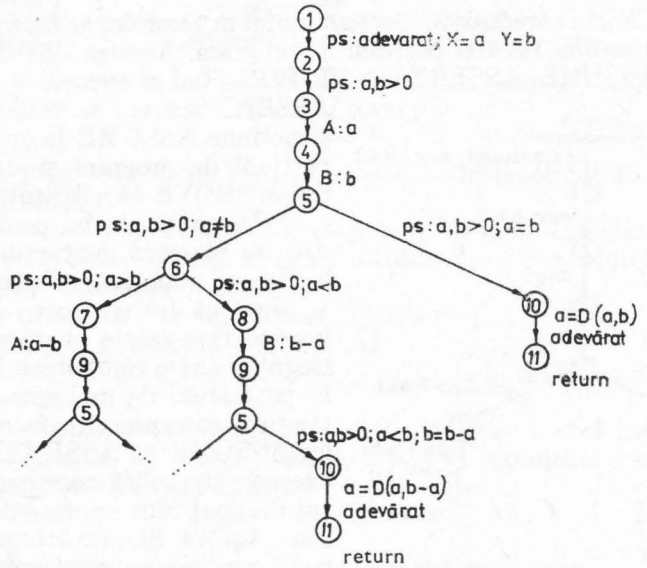


Fig. 7.2. — Arbore de execuţie simbolică pentru programul de calcul al celui mai mare divizor comun a două numere întregi pozitive.

Pentru a evita arborii de execuţie simbolică infiniţi se pot aplica două metode:

— fie se introduc aserţiuni invariante pentru traversarea ciclurilor;

— fie se introduce o restricţie în aserţiunea de intrare; de exemplu, pentru programul de mai jos dacă se introduce instrucţiunea ASSUME ($x > 0$ şi $y > 0$ şi $x = C * y$ şi $C \leqslant 1000$) se au în vedere numai arborii finiţi pentru care un argument este un multiplu de celălalt. Această procedură

restrînge clasa de date de intrare pentru program, de aceea, în general, se preferă prima.

Adăugăm în programul precedent o aserțiune intermediară în bucla WHILE:

```
            1. DIVIZOR: PROCEDURE (X, Y);
tăietura 1  2.     ASSUME (X > 0 și Y > 0);
            3.     A = X;
            4.     B = Y;
            5.     WHILE (A ≠ B) DØ;
tăietura 2  6.         ASSERT (CMMDC(A,B) = CMMDC(X,Y) și A ≠ B);
            7.         IF  A > B
            8.             THEN A = A – B
            9.             ELSE B = B – A
           10.     END;
           11.     PRØVE (A = CMMDC(X, Y));
           12.     RETURN (A);
           13. END
```

Introducerea aserțiunii intermediare ASSERT, împreună cu instrucțiunile ASSUME și PRØVE, decupează programul în mai multe porțiuni și se face o execuție simbolică separată pentru fiecare porțiune. Bucla fiind decupată va avea o execuție simbolică finită.

Demonstrarea corectitudinii programului în acest caz se face prin demonstrarea corectitudinii fiecărei porțiuni de program. Acestea sînt delimitate de instrucțiuni ASSUME, ASSERT sau PRØVE. Cînd se execută o instrucțiune ASSERT aceasta se tratează ca o instrucțiune ASSUME la începutul unei porțiuni de program și ca o instrucțiune PRØVE la sfîrșitul ei.

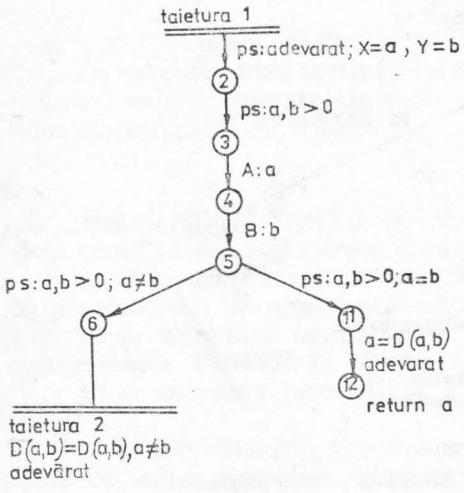


Fig. 7.3. — Arborele 1 de execuție simbolică corespunzător instrucțiunilor ASSUME și ASSERT.

În concluzie se poate spune că dacă se plasează instrucțiuni ASSERT la fiecare „tăietură" a programului și se arată că fiecare parte este corectă în raport cu aceste aserțiuni, atunci întregul program este corect. De exemplu, în programul de mai sus există două tăieturi, corespunzătoare instrucțiunilor ASSUME și ASSERT. Arborii de execuție simbolică corespunzători acestui decupaj sînt reprezentați în figurile 7.3 și 7.4. Fiecare arbore se termină pe o condiție ce este verificată, ceea ce înseamnă că programul este corect.

*

Metoda prezentată mai sus a fost elaborată mai întîi de Deutsch [42] și dezvoltată apoi de King [41]. Implementarea unui verificator automat bazat pe execuția simbolică a programului se apropie de un interpretor de limbaj.

Problemele principale care apar în cazul execuției simbolice sînt legate de dificultățile de verificare ale expresiilor de tip PRØVE care sînt în genera foarte complexe. De asemenea trebuie găsite aserțiunile intermediare ce asi-

gură decupajul programului în porțiuni cu arbori finiți, ceea ce nu este deloc ușor.

S-au scris deja mai multe sisteme interactive de verificare a corectitudinii programelor pe baza execuției simbolice. În [39] se prezintă un astfel de sistem

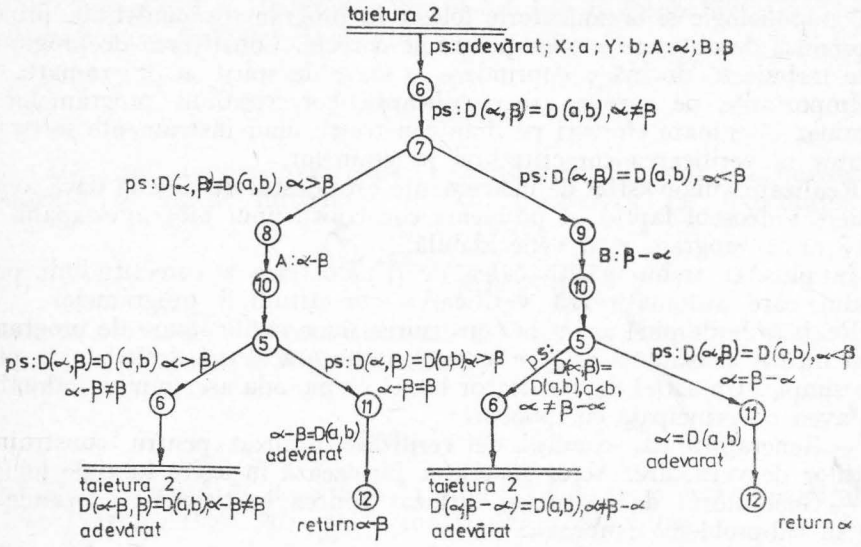


Fig. 7.4. — Arborele 2 de execuție simbolică corespunzător instrucțiunilor ASSUME și ASSERT.

care acceptă programe scrise într-un subset de instrucțiuni PL/1 și asigură următoarele facilități:

— utilizatorul poate lucra în același timp cu date simbolice și concrete;

— la instrucțiuni de tip IF nerezolvabile se dă controlul utilizatorului care decide selectarea ramurii:

— „go true": se execută blocul THEN;

— „go false": se execută blocul ELSE;

— „assume p; go": se evaluează p și se selectează ramura conform rezultatului obținut;

— utilizatorul poate salva starea sistemului (contextul înainte de execuția unei instrucțiuni IF), să exploreze o cale și să restaureze contextul inițial pentru parcurgerea celeilalte ramuri.

7.2.5. Concluzie privind verificarea corectitudinii programelor

Din cele prezentate în paragrafele precedente se poate spune că problema elaborării de programe corecte este deosebit de complexă. Este prin urmare esențial să se evidențieze, în completitudinea lor, aserțiunile invariante care implică furnizarea unor rezultate corecte. Este bine ca aceste aserțiuni să fie cunoscute înainte de a începe elaborarea programului.

Dacă în plus pe parcursul elaborării programului se pune condiția ca aserțiunile să fie tot timpul satisfăcute (adică în toate etapele elaborării lui), atunci programul va fi corect relativ la aserțiunile alese. Există și în acest caz o probabilitate de incorectitudine relativă la intenția programatorului (care

ține de posibilitatea strecurării unor erori în formularea aserțiunilor invariante sau de omiterea unor detalii esențiale tocmai din cauza urmăririi îndeosebi a aserțiunilor). Pentru că este greu să verificăm corectitudinea programelor este bine să așezăm la baza întregului proces de elaborare a programelor metode, tehnici și instrumente capabile să ducă la obținerea de programe corecte. Cadrul metodologic și organizatoric folosit în programare constituie principala premisă de obținere a unor programe corecte. Construirea de programe corecte trebuie să devină o deprindere, o stare de spirit în programare.

Importanța pe care o are verificarea corectitudinii programelor a determinat însemnate eforturi pe linia construirii unor instrumente software de ajutor în verificarea corectitudinii programelor.

Realizarea unor astfel de instrumente este destul de dificilă dacă avem în vedere îndeosebi faptul că problema construirii unui program capabil să verifice orice program este nedecidabilă.

În plus ar trebui găsită calea de demonstrare a corectitudinii programului care automatizează verificarea corectitudinii programelor.

Realizările de pînă acum în construirea unor verificatoare de programe nu sînt încă în măsură să asigure operaționalitate acestora decît pentru programe simple. Un astfel de verificator bazat pe metoda aserțiunilor inductive poate avea ca principale componente:

— Generatorul de condiții de verificare, utilizat pentru construirea condițiilor de verificare. Acest generator furnizează în ieșire formule logice.

— Generatorul de subcazuri, utilizat pentru partiționarea formulelor logice în subprobleme (subcazuri).

— Rezolvantul de subcazuri, utilizat pentru rezolvarea subcazurilor furnizate de generatorul de subcazuri folosind pentru aceasta un set de formule logice (leme) introduse de utilizator.

Un astfel de program va putea analiza programe scrise într-un limbaj de programare de nivel înalt însoțite de aserțiuni de intrare și respectiv de ieșire, precum și de aserțiuni inductive în diferite puncte ale programului.

7.3. Fiabilitatea programelor

7.3.1. Parametrii fiabilității elementelor

În sensul cel mai larg fiabilitatea reprezintă proprietatea unui dispozitiv de a-și păstra parametrii de ieșire în limitele admise, în condiții de exploatare date. Prin parametru de fiabilitate se înțelege o măsură cu ajutorul căreia se exprimă cantitativ fiabilitatea sau una din caracteristicile sale [51]. Se pot enumera foarte mulți parametri de fiabilitate, dar nici unul nu poate măsura complet fiabilitatea, ci doar estimează una din caracteristicile acesteia. Cel mai adesea, pentru dispozitivele hardware se consideră drept măsură principală a fiabilității probabilitatea funcționării fără erori în decursul unui interval de timp dat

$$R(t) = P(T > t),$$

unde T este variabila aleatoare a timpului de funcționare fără erori. Funcția $R(t)$ se mai numește funcția fiabilității dispozitivului. Uneori se utilizează probabilitatea ca eroarea să se producă în intervalul de timp t

$$F(t) = P(T \leqslant t),$$

F este chiar funcția de repartiție a timpului T de funcționare fără defecțiuni. $R(t)$ și $F(t)$ sînt evenimente contrarii, deci

$$R(t) = 1 - F(t).$$

Un alt parametru de fiabilitate este densitatea de probabilitate a timpului de funcționare fără defecțiuni $f(t)$

$$f(t) = \frac{\mathrm{d}F(t)}{\mathrm{d}t} = -\frac{\mathrm{d}R(t)}{\mathrm{d}t}.$$

Dacă se face presupunerea că aparițiile erorilor sînt evenimente independente se poate introduce un alt parametru de fiabilitate $\lambda(t)$

$$\lambda(t) = -\frac{1}{R(t)} \cdot \frac{\mathrm{d}R(t)}{\mathrm{d}t}$$

cunoscut sub numele de rata (intensitatea) erorilor. Rezolvînd relația de mai sus, și ținînd cont că $R(0) = 1$, rezultă

$$R(t) = \mathrm{e}^{-\int_0^t \lambda(t)\,\mathrm{d}t}.$$

Deci numărul de defecțiuni în intervalul de timp este repartizat după o lege Poisson. Dacă $\lambda(t)$ este o funcție constantă atunci

$$R(t) = \mathrm{e}^{-\lambda t}$$

$$F(t) = 1 - \mathrm{e}^{-\lambda t} = 1 - [1 - \lambda t + \ldots] \approx \lambda t.$$

Deci

$$\lambda \approx \frac{F(t)}{t}.$$

Aceasta înseamnă că se poate aproxima intensitatea erorilor prin numărul de erori (în procente) raportat la timpul de funcționare.

Frecvența medie a erorilor este un parametru asociat intervalului de timp $(0, t)$

$$\nu(t) = \frac{1}{t}\int_0^t f(t)\,\mathrm{d}t.$$

În cazul legii exponențiale, cu λ constant, avem

$$\nu(t) = \frac{1}{t}\int_0^t \lambda \mathrm{e}^{-\lambda t}\,\mathrm{d}t = \frac{1 - \mathrm{e}^{-\lambda t}}{t}.$$

În prima perioadă de funcționare a dispozitivului, cînd $\lambda t \ll 1$, deci $\mathrm{e}^{-\lambda t} \approx 1 - \lambda t$, rezultă

$$\nu(t) \approx \lambda.$$

Fiabilitatea se poate aprecia și prin timpul mediu de funcționare fără erori, T_0

$$T_0 = \int_0^\infty t f(t)\,\mathrm{d}t.$$

În cazul legii exponențiale, cu λ constant, obținem

$$T_0 = \int_0^\infty \lambda t e^{-\lambda t} \, dt = \frac{1}{\lambda}.$$

Drept parametru de fiabilitate se mai utilizează și termenul de garanție t_g. Dacă probabilitatea funcționării fără erori în perioada de timp t_g este cunoscută atunci din relația

$$R(t_g) = a$$

se determină termenul de garanție t_g.

În cazul dispozitivelor hardware studiul fiabilității are ca bază o teorie bine fundamentată cu largi aplicații practice. Fiabilitatea componentelor electronice a devenit o condiție tehnică, un parametru în proiectarea, fabricarea și exploatarea lor și una din problemele fundamentale ale tehnicii actuale.

Prin analogie cu fiabilitatea hardware au fost elaborate modele matematice ale fiabilității software, care caută să măsoare probabilitatea ca un program să funcționeze fără erori o perioadă dată de timp. Calculul acestei probabilități nu este însă evident; el se bazează pe mai multe ipoteze care nu corespund întotdeauna fenomenului destul de complex al introducerii și apariției erorilor software. De aceea fiabilitatea programelor se exprimă în general prin indicatori de fiabilitate (numărul de erori pe unitatea de timp, timpul mediu între două erori etc.) care rezultă din numărul de erori detectate în perioada de testare a produsului, înainte de livrare la beneficiar.

7.3.2. Abordarea empirică a fiabilității software

Din punct de vedere empiric se pune problema de a selecta anumite caracteristici sau atribute măsurabile ale programului, care împreună constituie un indicator asupra fiabilității lui. Nu este suficient de a defini atributele legate de fiabilitatea programului, ci trebuie indicată și o metodă de măsurare sau evaluare a lor. Astfel de atribute pot fi: claritatea programului, ușurința în testare, ușurința în întreținere etc.

Un model al studiului empiric al fiabilității îl constituie modelul complexității [52]. El se bazează pe faptul că un mare număr de erori software se explică prin atributul de complexitate al programului.

Se consideră următoarele categorii de complexitate:

a) *Complexitatea logicii* se referă la codul sursă al programului, adică număr de instrucțiuni executabile, număr de instrucțiuni de selecție, număr de instrucțiuni de iterație etc. Se măsoară astfel

$$C_{log} = \frac{I_t}{I_e} + C_{it} + C_{sel} + C_{salt},$$

unde I_t este numărul total de instrucțiuni, I_e — numărul de instrucțiuni executabile, C_{it} — complexitatea iterațiilor; se măsoară astfel

$$C_{it} = \sum_i m_i w_i,$$

unde m_i este numărul de cicluri de nivel i, w_i — factor de ponderare, și anume

$$w_i = 4^{i-1} \cdot \frac{3}{4^Q - 1}, \text{ astfel încît } \sum_{i=1}^{Q} w_i = 1,$$

Q fiind numărul maxim de imbricări de cicluri în program.

C_{sel} este complexitatea instrucțiunilor de selecție (IF)

$$C_{sel} = \sum_i n_i w_i,$$

unde n_i este numărul de instrucțiuni IF de nivel i, w_i — factor de ponderare (vezi mai sus).

C_{salt} este numărul de instrucțiuni de salt necondiționat.

b) *Complexitatea interferențelor* se măsoară prin numărul de apeluri de rutine utilizator și rutine sistem

$$C_{int} = u + s/2,$$

unde u este numărul de apeluri de rutine utilizator, s — numărul de apeluri de rutine sistem.

c) *Complexitatea calculului* se măsoară prin numărul de instrucțiuni de asignare de valori ce conțin operatori aritmetici

$$C_{calc} = \frac{c}{I_e} \cdot \frac{C_{rut}}{\sum_i c_i} \cdot c,$$

unde c este numărul de instrucțiuni de calcul, I_e — numărul de instrucțiuni executabile, $C_{rut} = \sum_i C_{log/rut}$, suma complexității logicii tuturor rutinelor, $\sum_i c_i$ — suma instrucțiunilor de calcul ale tuturor rutinelor.

d) *Complexitatea intrării-ieșirii* se măsoară prin numărul de instrucțiuni de intrare-ieșire

$$C_{i-e} = \frac{I_{i-e}}{I_e} \cdot \frac{C_{rut}}{\sum_i i_{i-e}} \cdot I_{i-e},$$

unde I_{i-e} este numărul de instrucțiuni de intrare-ieșire, $C_{rut} = \sum_i C_{log/rut}$ — suma complexității logicii tuturor rutinelor, $\sum_i i_{i-e}$ — suma instrucțiunilor de intrare-ieșire din toate rutinele.

e) *Claritatea* se măsoară prin raportul dintre numărul de instrucțiuni cu comentariu și numărul total de instrucțiuni.

$$L = I_{com}/I_t,$$

unde I_{com} este numărul de instrucțiuni cu comentariu, I_t — numărul total de instrucțiuni.

Complexitatea totală se definește astfel

$$C = C_{log} + 0{,}1\, C_{int} + 0{,}2\, C_{calc} + 0{,}4\, C_{i-e} - 0{,}1L.$$

În cazul unei complexități ridicate se consideră că fiabilitatea programului este scăzută. În concluzie, modelul empiric al complexității calculează acei parametri, care dacă se reduc rezultă o îmbunătățire a fiabilității programului.

7.3.3. Estimarea fiabilității software prin modelul lui Shooman

Termenul de „model de fiabilitate software" se referă la un model matematic ce se construiește cu scopul de a determina fiabilitatea software cu ajutorul unor parametri ce se măsoară prin observații experimentale. De asemenea se poate referi la relațiile matematice dintre parametrii caracteristici fiabilității.

De exemplu, relația dintre drumurile logice din program și seturile de date de intrare ce determină parcurgerea acestor drumuri este relevantă pentru fiabilitatea programului. Frecvența erorilor este un alt parametru ce se referă indirect la fiabilitatea software. Modelul lui Shooman [53] estimează fiabilitatea programului pe baza numărului de erori detectate în perioada de depanare. El presupune următoarele:

a) După faza de integrare există N erori în program. Urmează o perioadă τ de depanare în cursul căreia se elimină $\alpha_c(\tau)$ erori pe numărul de instrucțiuni. Deci, numărul de erori rămase (pe număr de instrucțiuni), după perioada τ de depanare este

$$\alpha_r(\tau) = N/I - \alpha_c(\tau),$$

unde I este numărul total de instrucțiuni (constant).

b) Se presupune că rata erorilor $\lambda(t)$ este proporțională cu numărul de erori rămase după perioada τ de depanare

$$\lambda(t) = k\alpha_r(\tau).$$

Dacă t se socotește de la momentul $t = 0$ iar τ rămîne fix, atunci intensitatea erorilor este constantă, deci probabilitatea ca să nu apară nici o eroare în intervalul $(0, t)$ este

$$R(t, \tau) = e^{-k(N/I - \alpha_c(\tau))t}.$$

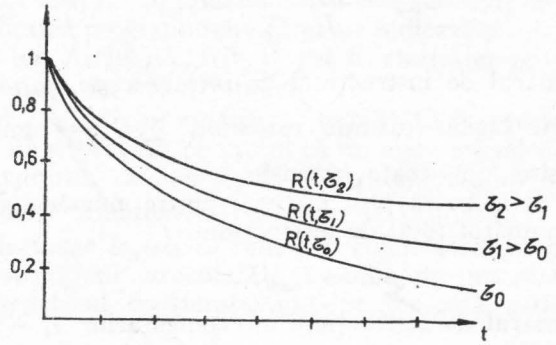


Fig. 7.5. — Fiabilitatea programului în funcție de timpul de depanare.

Expresia de mai sus exprimă formula de calcul a fiabilității programului conform modelului lui Shooman. Ea are ca parametru timpul de depanare τ. Dacă se reprezintă grafic fiabilitatea programului în funcție de τ se obțin curbele din figura 7.5, care ilustrează faptul că fiabilitatea programului crește dacă perioada de depanare este mai îndelungată.

Timpul mediu T_0 de funcționare între erori este

$$T_0(\tau) = \int_0^\infty R(t, \tau)\, dt = \frac{1}{k(N/I - \alpha_c(\tau))}.$$

Presupunem o rată constantă de corectări de erori, deci

$$\alpha_c(\tau) = \rho\tau.$$

În acest caz T_0 se poate scrie

$$T_0(\tau) = \frac{1}{\beta(1 - \mu\tau)}.$$

Dacă se reprezintă $T_0\beta$ în funcție de $\mu\tau$ se obține un grafic ca cel din figura 7.6.

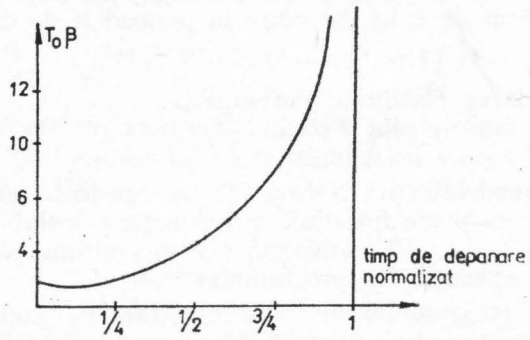


Fig. 7.6. — Grafic de exprimare a fiabilității.

Pentru calculul fiabilității programului utilizînd formula expresiei lui $R(t, \tau)$ trebuie estimate valorile necunoscutelor N și K ($\alpha_c(\tau)$ se obține prin înregistrarea erorilor eliminate în perioada de depanare τ). Valorile lui N și K se pot aproxima rulînd un program după două perioade de depanare diferite τ_1 și τ_2, astfel încît $\alpha_c(\tau_1) < \alpha_c(\tau_2)$.

La τ_1 avem

$$T_{01} = \frac{1}{\lambda_1} = \frac{1}{K(N/I - \alpha_c(\tau_1))}.$$

La τ_2 avem

$$T_{02} = \frac{1}{\lambda_2} = \frac{1}{K(N/I - \alpha_c(\tau_2))}.$$

Se face raportul și se obține

$$\widehat{N} = \frac{I[(\lambda_2/\lambda_1)\, \alpha_c(\tau_1) - \alpha_c(\tau_2)]}{(\lambda_2/\lambda_1) - 1}.$$

Din expresia lui T_{01} se scoate

$$\widehat{K} = \frac{\lambda_1}{\widehat{N}/I - \alpha_c(\tau_1)}.$$

Valorile λ_1 și λ_2 se aproximează astfel: după o perioadă τ_1 de depanare se execută programul de n_1 ori (cu n_1 seturi de date de intrare diferite). Dacă r_1 execuții se termină cu succes, rezultă $T_1, T_2, \ldots, T_{r_1}$ perioade de execuții fără erori. Pentru cele $n_1 - r_1$ execuții terminate pe eroare rezultă $t_1, t_2, \ldots, t_{n_1 - r_1}$ perioade de timp pînă la apariția erorii. Timpul total de execuții este

$$H_1 = \sum_{i=1}^{r_1} T_i + \sum_{i=1}^{n_1 - r_1} t_i.$$

Presupunînd rata erorilor constantă, atunci:

$$\lambda_1 = \frac{n_1 - r_1}{H_1}.$$

Analog se calculează λ_2 pentru perioada τ_2 de depanare. În concluzie, modulul lui Shooman stabilește o formulă de evaluare a fiabilității programului ce se bazează pe numărul de erori detectate în perioadele de depanare.

7.3.4. Estimarea fiabilității software prin modelul lui Nelson

În cadrul modelului lui Nelson [52] se consideră următoarele:

a) un program poate fi definit ca o funcție calculabilă F pe mulțimea $E = \{E_i \mid i = 1, 2, \ldots, N\}$, unde E_i este o mulțime de valori (set) de intrare pentru o execuție i a programului;

b) execuția programului cu setul de intrare E_i produce $F(E_i)$;

c) fie F' funcția implementată de program (datorită erorilor) și fie Δ_i toleranța rezultatului la execuția numărului i

$$|F'(E_i) - F(E_i)| \leqslant \Delta_i;$$

d) mulțimea E_e formaă din seturile E_i pentru care

$$|F'(E_i) - F(E_i)| > \Delta_{i_i}$$

corespunde la execuții eronate ale programului.

Fie n_e numărul de seturi E_i din E_e. Atunci probabilitatea ca execuția programului să se termine pe eroare este

$$P = n_e/N.$$

Probabilitatea ca execuția programului să se termine cu succes (funcția de fiabilitate) este

$$R = 1 - P = 1 - n_e/N.$$

Presupunem că seturile E_i de date de intrare nu au aceeași probabilitate de selectare și fie p_i probabilitatea ca programul să se execute cu setul de intrare E_i. În acest caz F se poate exprima în funcție de p_i dacă se introduce o variabilă de execuție y_i, ce ia valoare 0 dacă execuția programului cu setul de intrare E_i se termină cu succes, sau valoarea 1 dacă execuția programului cu setul de intrare E_i se termină pe eroare

$$P = \sum_{i=1}^{N} p_i y_i.$$

Atunci

$$R = 1 - \sum_{i=1}^{N} p_i y_i.$$

Dacă programul se execută de n ori probabilitatea ca cele n execuții să se termine cu succes este

$$R(n) = R^n = (1 - P)^n.$$

Fie p_{ji} probabilitatea ca setul E_i să fie selectat la execuția numărul j. Atunci probabilitatea ca execuția numărul j a programului să se termine pe eroare este

$$P_j = \sum_{i=1}^{n} p_{ji} y_i.$$

Probabilitatea să nu apară nici o eroare în cele n execuții este

$$R(n) = (1 - P_1)(1 - P_2) \dots (1 - P_n) = \prod_{j=1}^{n} (1 - P_j).$$

Deci, conform acestui model fiabilitatea programului se definește ca probabilitatea ca n execuții ale lui să se termine cu succes. Formula precedentă se mai poate scrie

$$R(n) = e^{\sum_{j=1}^{n} \ln (1-P_j)}.$$

Dacă $P_j \ll 1$ atunci $R(n) = e^{-\sum_{j=1}^{n} P_j}$,
Dacă $P_j \ll 1$ și $P_j = P =$ constant, atunci $R(n) = e^{-P \cdot n}$

Măsurarea fiabilității programului conform acestui model presupune determinarea probabilităților P_j. Se demonstrează [52] că o bună aproximare a lui R se poate obține rulînd programul cu n seturi de date de intrare și calculînd $\widehat{R}$ astfel

$$\widehat{R} = 1 - \hat{n}_e / n,$$

unde $\hat{n}_e$ este numărul de execuții terminate pe eroare. Se procedează astfel: se determină „profilul operațional al programului" stabilind probabilitățile p_i de selectare a seturilor E_i de date de intrare; se aleg n seturi de intrare corespunzătoare celor mai mari probabilități; se realizează cele n execuții; se contorizează execuțiile terminate pe eroare; se calculează $\widehat{R}$ cu formula de mai sus.

7.3.5. Evaluarea fiabilității software pe baza structurii programului

Metoda descrisă în [54] constă în estimarea exhaustivității testelor în scopul de a caracteriza fiabilitatea programului. Un program se consideră ca verificat dacă se poate dovedi că este corect pe mulțimea datelor sale de intrare. Pentru aceasta trebuie arătat că toate cazurile posibile de execuție

au fost testate. Acest lucru este însă imposibil în practică. Pe de altă parte un ansamblu arbitrar de teste nu este semnificativ pentru fiabilitatea programului. Selectarea unor date de test în funcție de structura programului poate deveni însă semnificativă. Pentru aceasta se reprezintă programul printr-un graf orientat, în care vîrfurile reprezintă instrucțiuni iar arcurile reprezintă legăturile posibile între instrucțiuni. Se va considera că un program este verificat dacă se poate dovedi că toate drumurile grafului programului au fost parcurse în timpul testării. Estimarea nivelului de testare necesită gruparea drumurilor logice din program în clase. În cazul acestei metode clasele sînt formate din drumuri ce ajung la o bifurcație. Scopul metodei este de a calcula numărul de drumuri ce pornesc de la punctul de intrare în program și ajung la fiecare bifurcație. Calculul se bazează pe faptul că programul este elaborat conform tehnicii programării structurate, adică toate drumurile ce ajung la intrarea unei structuri de control (sau mai general a unui bloc format dintr-o imbricare de structuri de control) vor cuprinde toate arcurile structurii de control. Acest lucru nu este posibil în cazul unor salturi anarhice în interiorul structurii de control.

Prezența ciclurilor face ca numărul drumurilor posibile la execuție să fie foarte mare. Acest număr se reduce limitîndu-se numărul de execuții al fiecărui corp de ciclu. Astfel, în cazul unui singur ciclu arcul de întoarcere (ciclare) nu va fi parcurs decît o singură dată în cadrul oricărui drum. Două drumuri, în acest caz, diferă unul de altul dacă: unul nu include arcul de ciclare, iar celălalt îl include; nici unul nu include arcul de ciclare dar cuprind arcuri diferite în cadrul ciclului; amîndouă includ arcul de ciclare dar cuprind arcuri diferite în ciclu. Exemplu în figura 7.7.

În cazul ciclurilor imbricate se aplică aceeași regulă de parcurgere a arcurilor de retur: o singură dată la fiecare nivel de ciclu.

Din figura 7.8 se observă că două parcurgeri ale arcului de intoarcere dintr-o buclă interioară sînt separate prin cel puțin o parcurgere a arcului de întoarcere a buclei exterioare. Pentru studiul drumurilor în graful programului sînt semnificative doar bifurcațiile. Pentru aceasta se reduce graful

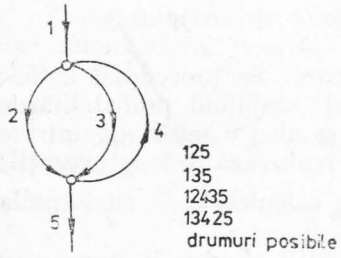


Fig. 7.7. — Exemplu de cicluri.

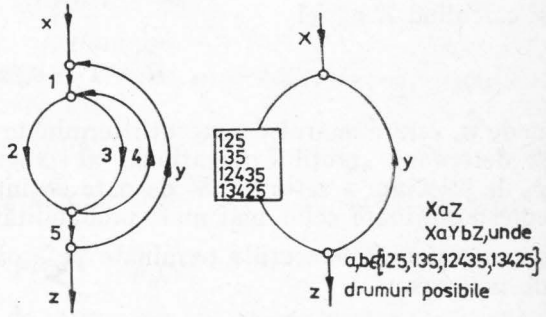


Fig. 7.8. — Parcurgerea ciclurilor.

programului astfel: se păstrează arcul de intrare în program, arcul de ieșire din program și nodurile de ramificație. Toate drumurile dintre nodurile de bifurcație se reprezintă printr-un singur arc. În cazul grafului redus se poate construi un algoritm simplu pentru calculul tuturor drumurilor din graf [54].

Ceea ce ne interesează este numărul drumurilor care ajung la o bifurcație. Fie subrutina următoare scrisă în limbajul FORTRAN:

```
1          SUBRØUTINE TEST(N)
2          DIMENSIØN TAB (10)
3          DØ 20 I = 1, N
4          J = I/2
5          IF (I.EQ.J) GØTØ  10
6          TAB(I) = 0
7          GØTØ 20
8     10   T(I) = I
9     20   CØNTINUE
10         RETURN
11         END
```

Graful programului este reprezentat în figura 7.9 *a*, iar în figura 7.9 *b* se reprezintă graful redus. Numărul drumurilor care ajung la fiecare bifurcație este reprezentat în figura 7.9 *c*.

Se construiește o tabelă care conține numărul $d(x, y)$ al tuturor drumurilor care ajung la bifurcația (x, y). De exemplu, pentru subrutina prezentată mai sus se construiește tabela din figura 7.10.

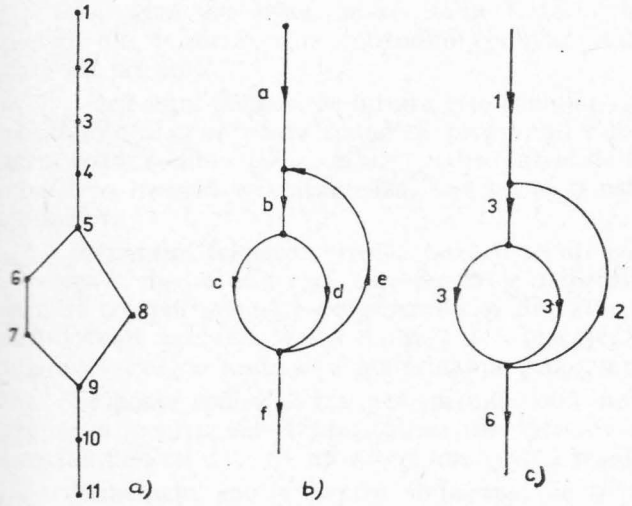


Fig. 7.9. — Graful programului.

(x,y)	d(x,y)
a b	1
b c	3
b d	3
c e	1
c f	3
d e	1
d f	3
e b	1

Fig. 7.10. — Tabelă pentru structură.

După construirea tabelei drumurilor, ce rezultă din analiza statică a programului, se trece la faza de execuție cu date experimentale de test în cursul căreia se înregistrează numărul de drumuri ce ajung la fiecare bifurcație. Fie $e(x, y)$ numărul de drumuri experimentale ce ajung la bifurcația (x, y). Nivelul de verificare al bifurcației (x, y) este dat de raportul

$$r(x, y) = \frac{e(x, y)}{d(x, y)}.$$

Se definește un nivel de verificare ponderat în funcție de constanta $k \geqslant 1$ astfel

$$v(x, y) = 1 \text{ dacă } \frac{e(x, y)}{\mathrm{d}(x, y)} \geqslant 1,$$

$$v(x, y) = \frac{e(x, y)}{K + d(x, y)} \text{ dacă } \frac{e(x, y)}{d(x, y)} < 1.$$

Se evaluează fiabilitatea programului prin expresia următoare

$$F = \frac{1}{N} \sum_{(x, y)} v(x, y),$$

unde N este numărul de bifurcații din program. Cu cît F este mai mare, cu atît nivelul de verificare al programului este mai ridicat, în sensul parcurgerii mai multor drumuri din program.

8. Testarea și depanarea programelor

8.1. Noțiuni de bază

După cum am mai menționat programele obținute prin codificare pot conține erori. Este deci necesară depistarea și eliminarea acestora.

Cum programele complexe se realizează pe module, fiecare modul trebuie *testat* mai întîi separat pentru a i se elimina erorile iar după aceea ca o componentă a unui program complex. Spunem că se procedează la *integrarea* modulului și *testarea de ansamblu* a programului complex (care în realitate se compune din mai multe programe ce comunică între ele pe bază de parametri, pe bază de fișiere etc.).

Erorile detectate în faza de testare trebuie eliminate. În limbajul uzual identificarea erorilor și corectarea programului este cunoscută și sub numele de *depanare*.

Așa cum am arătat în capitolul 7, testarea constituie totodată și o metodă de verificare a corectitudinii programului pentru un set limitat de date de intrare.

Dacă setul de date de intrare este complet, adică acoperă toate cazurile posibile, atunci se poate spune că programul este corect. Practic nu se pot testa toate combinațiile posibile pentru datele de intrare și de aceea testarea rămîne o metodă ce garantează, dar nu demonstrează, corectitudinea programelor.

Avantajul folosirii metodei testării rezultă din faptul că programul se execută în mediul său real, corespunzător utilizării lui finale și deci se poate urmări comportamentul programului și din alte puncte de vedere decît al rezultatelor propuse: timp de răspuns, blocaje, memorie ocupată etc. Se poate evalua de asemenea performanța programului.

Se poate spune că testarea include două etape: analiza structurii programului pentru selectarea datelor de intrare: testarea statică; execuția programului cu date de intrare și observarea rezultatelor: testarea dinamică.

Principalul motiv pentru elaborarea de tehnici de testare este costul tot mai ridicat al produselor program. Boehm [51] arată că, actual, costul fazei de testare a produselor program reprezintă aproximativ 48% din costul total de elaborare. Cu cît eroarea este detectată mai tîrziu în fazele de elaborare a produsului program cu atît costul eliminării ei este mai mare.

Alberts [52], stabilește un factor cu care trebuie multiplicat costul eliminării unei erori în diferite faze, și anume:

— concepere program	1
— testare modul	4
— testare funcție	9
— testare sistem	15
— întreținere sistem	30.

8.2. Strategii de testare

Cele mai cunoscute strategii de testare sînt: testarea de jos în sus (bottom-up); testarea de sus în jos (top-down); metoda mixtă.

Testarea de jos în sus este metoda clasică de testare, ea constă în testarea modulelor, urmată de testarea ansamblelor de module, urmată în final de testarea sistemului ca un tot unitar.

Testarea modulelor se face separat, pentru fiecare modul în parte, într-un mediu „artificial" care să asigure executarea funcțiilor modulului. Ea este cu atît mai ușor de efectuat cu cît modulele sînt mai simple și mai mici, iar funcțiile pe care le execută sînt relativ independente. În mod normal testarea modulelor trebuie să fie cît mai completă pentru a fi siguri de execuția corectă a funcțiilor acestora.

La fiecare modul trebuie făcută în primul rînd testarea de funcționalitate, adică trebuie verificat dacă modulul îndeplinește funcția sau funcțiile sale. Pentru aceasta se face o listă a funcțiilor modulului și se stabilesc date de test corespunzătoare ce pun în evidență comportarea modulului în condiții normale de lucru. De exemplu, pentru un modul de verificare sintactică a comenzilor utilizatorului se introduc ca date de intrare toate comenzile din lista de comenzi posibile. Se testează apoi posibilitatea de detectare de către modul a datelor eronate, ce nu corespund funcției sale. În cazul exemplului precedent se introduc comenzi voit eronate pentru a verifica dacă modulul reacționează cu mesajul de eroare adecvat.

În final se testează comportarea modulului în condiții „extreme" de lucru sau „la limită". De exemplu pentru un modul ce prelucrează datele dintr-un fișier se testează și următoarele cazuri: fișier vid, fișier cu un singur articol, fișier pe mai multe volume.

Testarea modulelor trebuie efectuată de o manieră sistematică, adică pentru fiecare modul se face o listă a testelor efectuate și a rezultatelor obținute. Aceasta facilitează procesul ulterior de depanare al modulelor.

Un subsistem este compus din mai multe module. Testarea subsistemului presupune verificarea interfețelor dintre module.

Sistemul este un ansamblu de mai multe subsisteme. Testarea sistemului înseamnă verificarea comunicării dintre subsisteme. În general testarea sistemului nu poate fi exhaustivă. Ea presupune verificarea sistemului din punct de vedere al specificațiilor sale, dar și al performanțelor: timp de răspuns, capacitate de lucru etc. Testele de sistem pot ocupa pînă la 30% din timpul total de realizare al proiectului. Dezavantajul testării de jos în sus rezultă din dificultatea de a testa modulele separat, deoarece trebuie simulată interacțiunea modulului cu alte module ce asigură datele de intrare corespunzătoare (fig. 8.1). Se apreciază că rutinele suplimentare de simulare pot reprezenta pînă la 50 % din cadrul modulului de testat, ceea ce reprezintă un efort în plus apreciabil.

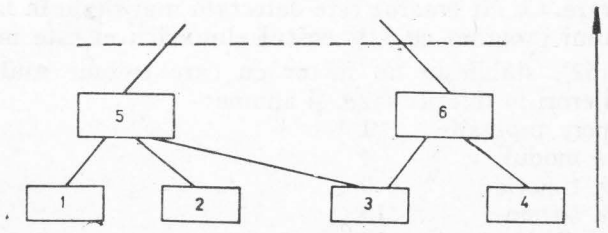


Fig. 8.1. — Testarea modulelor de jos în sus.

De asemenea, este foarte dificil de stabilit datele de test pentru sistemul final datorită numeroaselor subsisteme ce îl compun.

Erorile care pot să apară în acest caz sînt legate de modul de comunicare între module: date neutilizate, date inițializate în mai multe module, date modificate în diferite module etc.

Pentru o mai ușoară depanare testarea interconectării modulelor se poate face treptat urmărind procesul de elaborare al programului.

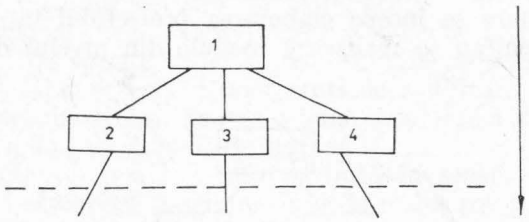


Fig. 8.2. — Testarea modulelor de sus în jos.

Testarea de sus în jos este posibilă numai în cazul unei conceperi descendente a sistemului. În acest caz se începe cu testarea modulului principal (de bază) și a primului nivel subordonat de rutine (fig. 8.2). Apoi, pe măsură ce avansează realizarea proiectului se continuă testarea sistemului obținut prin adăugarea nivelului al 2-lea ș.a.m.d. Pentru efectuarea testării vor fi utilizate module „fictive", din niveluri inferioare, nerealizate încă, dar apelate de nivelurile superioare. Modulele nerealizate încă, dar apelate de modulele din nivelurile superioare conțin doar instrucțiunea de revenire la apelant. Eventual pot tipări un mesaj că au fost apelate, sau pot furniza unele date care apar ca rezultatul unei prelucrări.

Interconectarea progresivă facilitează în mod apreciabil elaborarea sistemului final, erorile finale fiind foarte puține și ușor de depanat.

Multe erori de interconectare a modulelor pot fi eliminate înaintea procesului de testare dacă se construiește matricea parametrilor de interconectare. Pentru aceasta se face o listă a tuturor parametrilor de intrare/ieșire pentru toate modulele și o listă a modulelor. Se construiește apoi matricea M de interconectare, astfel

$$M(p_i, m_j) = x,$$

unde p_i este parametrul de intrare/ieșire numărul i; m_j este modulul numărul j; x poate fi: I — dacă parametrul p_i este inițializat în modulul m_j, U — dacă parametrul p_i este utilizat de modulul m_j, Z — dacă parametrul p_i este modificat de modulul m_j.

Odată construită matricea M se observă foarte ușor următoarele tipuri de erori: un parametru nu este inițializat; un parametru este transmis unui modul, dar neutilizat; un modul modifică un parametru și acest lucru nu este permis etc.

Avantajele testării de sus în jos sînt următoarele: testele de sistem se efectuează pe măsura elaborării lui, deci în final sînt reduse; se testează mai întîi interfața dintre module eliminîndu-se de la început erorile dificil de detectat; erorile sînt localizate în modulele noi adăugate, deci eliminarea lor este mult ușurată.

Prin efectuarea testării în paralel cu elaborarea proiectului, pe total, se reduce considerabil timpul de elaborare al unui produs-program (uneori cu aproape 1/3).

Metoda de testare mixtă presupune aplicarea simultană a celor două metode precedente de testare. Uneori nu este posibil de a utiliza module „fictive" necesare nivelelor superioare și atunci se impune realizarea și testarea separată a unor module din nivelele de jos ale ierarhiei. În acest caz metoda testării de sus în jos se împletește cu cea a testării de jos în sus, rămînînd predominantă testarea descendentă.

Prin urmare se începe elaborarea proiectului într-o manieră descendentă, dar simultan se realizează module din nivelul de bază al ierarhiei (fig. 8.3).

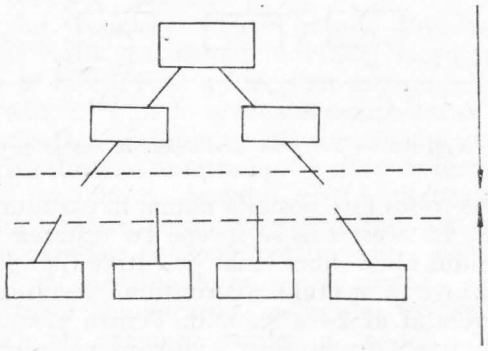


Fig. 8.3. — Testarea mixtă.

În acest caz testarea descendentă se desfășoară în paralel cu cea ascendentă. Acest procedeu de concepere și testare s-a dovedit eficient în elaborarea multor produse-program.

Alegerea strategiei de testare depinde de modulul de realizare al proiectului. Într-o abordare de realizare ascendentă, testarea de jos în sus este cea mai potrivită și reciproc testarea de sus în jos se aplică unui proiect realizat prin metoda descendentă. Metoda de testare de jos în sus se poate aplica practic pentru orice sistem indiferent de modul de realizare, cu observația că pentru sistemele elaborate prin metoda descendentă nu se poate aplica decît după obținerea sistemului final (spre deosebire de testarea descendentă, care în acest caz, urmărește realizarea proiectului).

8.3. Moduri de testare

Testarea poate fi: funcțională sau structurală.

Testarea funcțională (black-box) are drept scop verificarea specificațiilor modulului: nu se urmărește modul de realizare al modulului ci numai comportamentul lui exterior, adică felul în care își îndeplinește funcția sa. Datele de test se construiesc din specificațiile de definiție ale modulului. Modulul este privit ca o „cutie neagră", fără a avea acces la codul său intern, în care se introduc date de intrare și se obțin date de ieșire. Pe baza analizei datelor de intrare și ieșire se deduce dacă funcția modulului este realizată corect sau nu. Testarea funcțională se aplică mai ales sistemelor finale, a căror com-

plexitate nu permite realizarea de teste pe baza structurii interne şi deci nu se pot verifica decît pe baza specificaţiilor de definiţie.

Testarea structurală (white-box) presupune analiza modulului de realizare a programului şi selectarea datelor de test pe baza structurii lui interne. Prin setul de date de intrare se caută să se parcurgă toate drumurile grafului asociat programului, dacă aceasta este posibil, dacă nu, se testează drumurile critice ale programului. Acest mod de testare se aplică foarte uşor la nivel de modul şi permite detectarea erorilor de codificare ale modulului. În general, în practică se aplică mai mult testarea structurală pentru că programatorul cunoaşte în detaliu modulul pe care l-a elaborat.

Testarea funcţională are avantajul de a detecta implementarea incorectă (neconformă cu specificaţiile) a unor funcţii. De asemenea, testarea structurală are avantajul detectării unor erori de codificare a modulului ce nu ţin de definirea specificaţiilor. Pentru o bună detectare a erorilor este bine să se aplice simultan ambele moduri de testare.

Testarea structurală poate fi automatizată în sensul elaborării unor sisteme de analiză a structurii programelor şi de elaborare a datelor de test.

8.4. Metodologia testării structurate

Odată cu apariţia conceptelor ingineriei programării s-a dezvoltat o nouă metodologie a procesului de testare, cunoscută sub numele de metodologia testării structurate. Ea se bazează în special pe aplicarea principiilor programării structurate în activitatea de programare, principii ce includ procesul de testare în tot ciclul de elaborare al produselor program. Iniţial testarea era văzută ca o etapă distinctă după elaborarea produsului final, adică după ce totul a fost codificat. În acest caz detectarea unor erori introduse chiar în faza de concepere devine foarte costisitoare, impunînd modificări de cod, uneori radicale.

Metodologia testării structurate presupune efectuarea de teste, controale şi inspecţii pe tot parcursul de elaborare al produsului program: definire specificaţii, concepere, codificare. În acest fel testarea devine o parte integrantă a procesului de elaborare al programului. Metodologia testării structurate include următoarele tehnici de testare: analiza de concepere; testarea statică (analiza statică); testarea dinamică; generarea datelor de test.

8.4.1. Analiza de concepere

Orice program este elaborat pornind de la specificaţiile sale şi orice eroare introdusă la acest nivel se va reflecta în codul final. De aceea analiza de concepere trebuie să înceapă cu verificarea specificaţiilor proiectului.

Din analiza specificaţiilor se pot detecta erori referitoare la funcţiile programului (specificaţii eronate) sau erori de logică (specificaţii implementate incorect).

În practică specificaţiile sînt rareori prezentate de o manieră formală. De cele mai multe ori ele sînt incomplete, ambigue, vagi etc. şi foarte multe erori sînt generate de specificaţii incorecte.

Pentru elaborarea unor specificaţii corecte este necesar ca acestea să se prezinte într-o manieră formalizată în scopul de a putea fi analizate şi testate pentru validitatea lor.

Parnas [53] propune un sistem de prezentare formală a specificațiilor. Specificațiile se reprezintă într-o formă de procese de intrare/ieșire iar sistemul de specificat se consideră ca o secvență de procese ce interacționează. Fiecare proces este echivalent cu o mașină secvențială a cărei comportare este dată de relațiile dintre intrări, ieșiri, și stări. Procesele se specifică static prin expresii logice și algebrice ce permit verificarea și testarea lor.

Altă metodă de prezentare formală a specificațiilor este un limbaj special de descriere a lor. Un astfel de limbaj a fost dezvoltat în cadrul unei metodologii de control a specificațiilor. Limbajul RSL (Requirements Statement Language) [6] este un limbaj ce suportă instrucțiuni de definire a specificațiilor și informații de organizare a lor, iar sintaxa și semantica lui sînt neambigue și analizabile pe mașină. După verificarea specificațiilor se trece la revizuirea procesului de concepere în scopul de a detecta erori de funcționare înainte de a trece la codificarea propriu-zisă. Printr-o nouă parcurgere a specificațiilor de realizare conceptorul poate detecta multe erori datorate, de exemplu, unor presupuneri greșite. Revizuirea proiectului poate fi făcută de conceptor însăși, dar experiența arată că această reparcurgere a procesului de proiectare este mai eficientă dacă este realizată de o altă persoană din grupul de conceptori. Această inspecție nu are drept scop găsirea de noi soluții ci numai detectarea de erori în soluția prezentată. Revizuirea codului nu este o operație simplă, dar rezultatul ei este foarte important, multe erori a căror detectare după elaborarea proiectului ar fi foarte costisitoare sînt eliminate din prima fază a proiectului.

În ceea ce privește automatizarea procesului de verificare a fazei de concepere, Boehm [55] prezintă rezultatele obținute prin utilizarea unui verificator al Consistenței Procesului de Concepere (DACC).

Utilizatorul verificatorului trebuie să specifice pentru fiecare modul un antet ce cuprinde aserțiuni referitoare la:

- sursa fiecărei variabile de intrare;
- destinația fiecărei variabile de ieșire;
- dimensiunea fiecărei variabile;
- tipul fiecărei variabile;
- intervalul de valori al fiecărei variabile etc.

El a fost aplicat, de exemplu, unui sistem de 186 module cu 514 aserțiuni de intrare și 453 aserțiuni de ieșire. Verificatorul DACC a descoperit 7 erori în privința aserțiunilor, 25 de nepotriviri de terminologie și 200 de situații în care diferite nume erau utilizate incorect ca intrări și ieșiri.

8.4.2. Analiza statică

Analiza statică se referă la verificarea codului programului cu scopul de a detecta erori de structură, erori de sintaxă și de a urmări dacă au fost respectate construcțiile standard ale limbajului de codificare.

Analiza statică nu implică execuția programului. Ea se aseamănă cu procesul de compilare al programului, ambele urmărind analiza structurii codului. Informațiile ce se obțin în urma analizei statice pot fi: variabile neinițializate; apeluri de funcții sau subrutine cu parametri incorecți; variabile neutilizate; porțiuni de cod ce nu vor fi executate; porțiuni de cod redondante etc. Nu pot fi detectate erorile legate de execuția programului, de exemplu erorile de referire a elementelor unui tablou.

Analiza statică se efectuează prin simpla parcurgere a listingului programului. Evident această analiză este mult facilitată dacă programul este lisibil; acest lucru depinde de limbajul de programare utilizat și de tehnica de programare aplicată. Programele structurate cu comentarii și nume de variabile și subrutine ce reflectă funcția lor sînt mai ușor de verificat decît programele nestructurate. Parcurgerea codului poate fi făcută de programatorul elaborator sau de alt programator (Walk-throughs). De exemplu, pentru apelul unei subrutine se verifică dacă: numărul parametrilor este corespunzător; parametrii sînt transmiși corect; tipul parametrilor corespunde utilizării lor.

Pentru instrucțiuni de selectare de tipul:

IF X = O THEN DØ ... END

se verifică: dacă testul trebuie făcut pe variabila X; dacă blocul de instrucțiuni de executat (DØ ... END) corespunde condiției testate.

Se apreciază că 90% din totalul erorilor sintactice pot fi eliminate prin analiza codului. Analiza statică poate fi automatizată și există deja mai multe tipuri de analizatoare de cod. Unele analizează doar un singur modul separat, ignorînd relațiile cu alte module. Un exemplu de verificator de cod este SQLAB (Software Quality Laboratory), utilizat deja pe o mare varietate de sisteme de calculatoare: CDC 7600, IBM 360/75, Honeywell 6180, UNIVAC 1108 etc [6]. SQLAB analizează programe scrise în FORTRAN sau PASCAL, după ce acestea au fost deja corectate prin eliminarea erorilor de compilare. SQLAB detectează următoarele tipuri de erori: variabile utilizate neconform tipului lor; variabile neinițializate sau definite și neutilizate; apeluri de subrutine cu parametri incorecți; porțiuni de cod neexecutabile.

Pentru verificarea utilizării variabilelor conform tipului lor se definesc și se introduc aserțiuni speciale în cadrul programului.

Un alt sistem de verificare de cod este DAVE [57], dezvoltat la Universitatea din Colorado. Acest sistem verifică în principal dacă o variabilă a fost inițializată înainte de utilizarea ei. De asemenea verifică echivalențele implicate de declarațiile CØMMØN din diferitele subrutine.

În concluzie analiza de cod este o metodă foarte eficientă și economică de detectare a erorilor înainte de punerea în exploatare a programului final.

8.4.3. Testarea dinamică

Testarea dinamică înseamnă execuția programului cu diverse seturi de date de intrare și compararea rezultatelor cu cele așteptate.

Prin testarea dinamică se detectează erori de execuție ale programului, ce nu pot fi descoperite prin analiza statică.

Testarea dinamică cuprinde trei etape: pregătirea testelor; execuția testelor; evaluarea rezultatelor.

Pregătirea sau planificarea testelor se referă la stabilirea funcțiilor de testat, a datelor de testare corespunzătoare acestor funcții, eventual la inserarea unor instrucțiuni suplimentare de tipărire a unor rezultate parțiale și la stabilirea rezultatelor ce trebuie obținute.

Executarea testelor presupune introducerea datelor de intrare în sistemul de testat, execuția programului și înregistrarea rezultatelor.

Evaluarea rezultatelor se referă la compararea rezultatelor obținute în urma execuției programului cu cele stabilite în faza de pregătire a testelor.

În timpul testării dinamice se pot înregistra statistici asupra utilizării variabilelor, execuției anumitor secvențe de instrucțiuni sau a unor rutine etc. statistici ce pun în evidență comportarea programului pentru diferite seturi de date de intrare. Se pot trage astfel concluzii asupra caracteristicilor de performanță ale programului.

Stabilirea datelor de test nu este o operație ușoară. S-au experimentat diverse metode de selectare a datelor de test și anume:

— testarea drumurilor: datele de test se aleg în așa fel încît să se parcurgă cel puțin odată fiecare drum din graful programului;

— testarea condițiilor de selectare: fiecare condiție de selectare să fie traversată cel puțin odată.

— testarea funcțională fiecare funcție implementată de program să fie testată cel puțin odată.

Practica arată că testarea funcțională este cea mai eficace, în sensul că în raport cu celelalte moduri de testare detectează cel mai mare număr de erori din orice program.

Algerea datelor de testare trebuie făcută avînd în vedere deopotrivă funcțiile procedurilor care definesc programul analizat și relațiile care se stabilesc între programele aparținînd aceluiași domeniu de aplicație.

Spre exemplu, la testarea unui program care realizează controlul datelor și crearea unui fișier cu datele corecte, avînd ca suport discul magnetic, prin alegerea datelor de testare se va urmări totodată obținerea în ieșire a setului de date necesar testării altor programe care folosesc în intrare fișierul astfel obținut.

În baza relațiilor existente între programele prezentate în figura 8.4 rezultă că la alegerea datelor de testare pentru programul UP01 avem în vedere deopotrivă funcțiile programului UP01 și funcțiile programului UP02 care folosește în intrare ieșirile programului UP01.

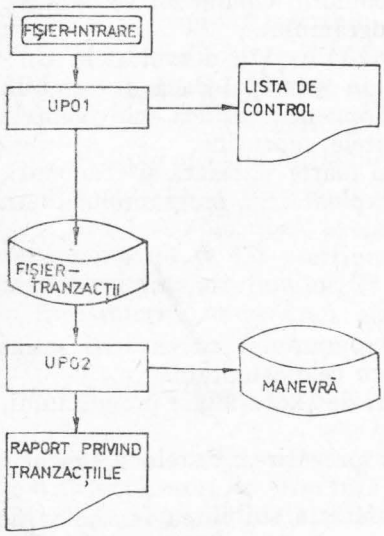


Fig. 8.4. — Relații între programe.

Pentru verificarea programului prin testarea dinamică se folosesc diverse procedee, ca:

— introducerea de instrucțiuni suplimentare pentru tipărirea unor rezultate intermediare sau avertizarea asupra execuției unei porțiuni de cod;

— introducerea unor „aserțiuni" prin care se verifică anumite rezultate în anumite puncte ale programului.

De exemplu, pentru a detecta cazul în care o variabilă *I* ia valori în afara intervalului [10 ... 50] se introduce aserțiunea

ASSERT (10 < = I AND I < = 50).

În cazul în care *I* ia valori în afara intervalului se tipărește un mesaj de avertizare anunțîndu-se apariția posibilă a unei erori.

Există compilatoare de limbaje de programare, ca PL/1 de exemplu, care prezintă o serie de facilități pentru detectarea erorilor de execuție ale programelor. De exemplu, compilatorul PL/1 generează cod pentru verificarea accesului la elementele unui tablou, adică elementul referit se află între limitele tabloului.

Fairly [58] propune o metodă interesantă de înregistrare a istoriei execuției programului pentru a fi analizată ulterior în caz de nefuncționare normală. Se înregistrează următoarele:

- execuția instrucțiunilor de asignare, selecție și GØTØ;
- apelul de proceduri cu valorile parametrilor de intrare și ieșire;
- valorile indecșilor de acces în structuri de tablou.

Metoda nu este practică pentru programele complexe din simplul motiv că datele de înregistrat devin numeroase.

Ca exemplu de instrument ajutător pentru obținerea și analiza rezultatelor testării dinamice se poate da sistemul PET (Program Evaleator and Tester) [59]. Acesta modifică programul în sensul completării lui (pre-procesing) în așa fel încît să culeagă cît mai multe date la execuția lui (post-procesing). Datele culese de PET cuprind:

— numărul de ori de execuție al unei instrucțiuni de asignare, împreună cu valoarea inițială, minimă, maximă și finală;

— numărul de ori de execuție a unei instrucțiuni de selecție și numărul de ori în care condiția a fost „adevărată" sau „falsă".

— numărul de ori de apel al unei subrutine etc.

Sistemul PET este util atît pentru detectarea erorilor cît și pentru obținerea unor statistici asupra utilizării diferitelor porțiuni de cod din program.

Sistemele de testare dinamică bazate pe aserțiuni examinează fiecare instrucțiune a programului și pe baza lor prelucrează condițiile din cadrul aserțiunilor. Aserțiunile pot fi active sau pe post de comentariu. În cazul în care sînt active ele sînt prelucrate și se tipărește un mesaj de avertizare în cazul în care rezultatul evaluării nu este adevărat. Dacă sînt pe post de comentariu atunci sînt ignorate de sistemul de testare. Ca exemplu de sistem de testare dinamică pe bază de aserțiuni se poate da SQLAB (Software Quality Laboratory) care încorporează și un procesor de analiză statică a programului. Rezultatele evaluărilor aserțiunilor se înregistrează într-un fișier ce poate fi consultat după terminarea execuției programului.

8.4.4. Generarea datelor de test

Problema generării datelor de test este următoarea: fiind dată o porțiune de cod dintr-un program să se genereze datele de test care provoacă la execuția programului și execuția acelei porțiuni de cod pe toate ramurile sale de program și în toate combinațiile dorite. Problema nu este simplă: practic nu există algoritm care să genereze date de test pentru atingerea unei porțiuni de cod dintr-un program. De aceea problema generării datelor de test se reduce la una din următoarele situații:

— *Testarea exhaustivă:* se propune testarea programului pentru toate datele de intrare posibile. Acest lucru nu este posibil decît pentru programe simple. Același termen se folosește și pentru cazul în care se încearcă parcurgerea fiecărui drum din program cel puțin o singură dată. Acest lucru este de asemenea greu de realizat. De exemplu o buclă care conține 5 drumuri și este parcursă de 100 de ori posedă 5^{100} secvențe diferite.

— *Testarea selectivă:* datele de test se selectează de către conceptorul programului care îi cunoaște în detaliu structura și funcțiile sale.

Se cunosc două tehnici mai importante pentru generarea datelor de test: generarea de date de test pentru parcurgerea drumurilor; generarea de date de test aleatoare.

În cazul primei tehnici se selectează mai întîi drumurile din program ce trebuie testate. Această selectare se poate face manual, de către programator, sau automat pe baza grafului programului.

Etapa următoare este generarea datelor de test pentru parcurgerea drumurilor selectate precedent. Acestea se realizează pe baza condițiilor de parcurgere a drumurilor. De exemplu, să presupunem că un drum conține secvența de instruciuni i_1, i_2, i_3. Dacă i_1 se execută cu condiția $p \geqslant 0$, i_2 cu condiția $q \leqslant 0$ iar i_3 cu condiția $p + q \geqslant 0$, atunci pentru parcurgerea secvenței $i_1 i_2 i_3$ trebuie alese date de intrare care satisfac condițiile precedente.

În cazul datelor de test aleatoare se aleg în mod arbitrar diverse date din domeniul variabilelor de intrare. Metoda este simplă și de cele mai multe ori testele efectuate sînt eficiente.

În literatura de specialitate există descrise o serie de generatoare automate de date de test (ATTEST, GENDATA etc.).

Un astfel de generator poate cuprinde trei părți principale: selectarea drumurilor; execuția simbolică; generarea datelor de test.

Pentru selectarea drumurilor de testat pot exista mai multe criterii dintre care enunțăm: selectarea tuturor drumurilor, ce se aplică pentru programele la care se dorește o testare exhaustivă; parcurgerea tuturor condițiilor de selectare; parcurgerea tuturor instrucțiunilor.

În funcție de criteriul ales se stabilesc drumurile de testat.

Analiza simbolică are drept scop stabilirea condițiilor de parcurgere pentru fiecare drum. De exemplu, în figura 8.5 condiția pentru parcurgerea drumului format din nodurile 1, 2, 4, 5 este: $a \geqslant 5$ și $b \leqslant 0$ și $(a + b) \leqslant 0$.

Generarea datelor de test se poate face pe baza condițiilor stabilite de analiza simbolică: se aleg date de intrare care satisfac aceste condiții. Datele de test rezultă deci ca soluții ale unui sistem de ecuații liniare corespunzătoare drumului de parcurs. De exemplu, dacă $a = 5$, $b = -5$ se parcurge drumul 1, 2, 4, 5 deoarece a și b sînt soluții ale sistemului de ecuații de mai sus. Drumul 1, 7, 8 are ca sistem de ecuații $a < 5$ și $c > 0$. Pentru $a = 4$ și $c = 1$ se realizează o parcurgere a acestui drum.

Sînt programe de generare a datelor de test care evită calculul apriori al drumurilor din program. Procedeul este următorul: se pornește de la un set inițial de date de test (ales arbitrar) și se determină drumurile parcurse la execuția programului cu acest set de date. Se face o execuție simbolică corespunzătoare acestor drumuri și se determină condițiile de parcurgere a acestora. Următorul set de date de test se stabilește pornind de la precedentul modificîndu-se cel puțin o dată de test pentru a nu satisface cel puțin o condiție din setul de drumuri determinat precedent. În felul acesta la execuția programului cu noul set de date de test se vor parcurge în mod obligatoriu și alte drumuri din program. Se continuă în felul acesta pînă se consideră că s-a ajuns la un nivel satisfăcător de testare al programului.

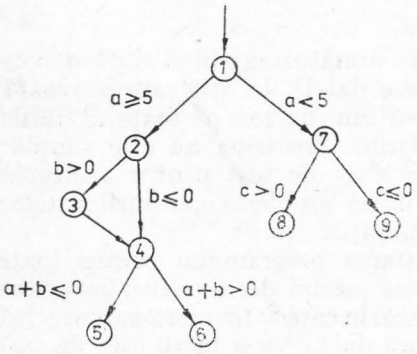


Fig. 8.5. — Exemplu pentru stabilirea drumurilor de testat.

În ceea ce privește modificarea programului ca urmare a depistării unor erori facem mențiunea că practica arată că probabilitatea de a modifica

în mod corect un program este apropiată de 50%. De aceea trebuie făcută o nouă testare a programului după orice modificare chiar minoră.

*

Se știe că prin testare nu se garantează corectitudinea programului. Nu există încă nici o metodă de testare care să demonstreze că programul este fără erori. Testarea asigură pe programator că programul său funcționează corect pentru un set limitat de date de intrare. Această garanție depinde de felul în care au fost alese datele de intrare pentru testare de către programator.

Lipsa unei metode riguroase care să asigure validarea produselor program constituie încă o problemă majoră a informaticii.

Aspectele teoretice elaborate pentru măsurarea eficacității testării nu sînt practice. Evaluarea testării se face în mod empiric în general pe baza unor rezultate statistico-experimentale.

Practica arată că analiza statică completată cu testarea dinamică conduce la programe cu un grad ridicat de funcționalitate. De asemenea, experiența arată că este bine ca validarea unui program să se realizeze prin mai multe metode de testare. Pe de altă parte, testarea programelor se simplifică dacă acestea se construiesc după o metodologie sistematică bine documentată.

8.5. Depanarea programelor

8.5.1. Categorii de erori

În urma proiectării și codificării se obțin programe care nu sînt neapărat și corecte. Prin depanarea unui program se urmărește localizarea zonelor din program care au dus la apariția de erori, analiza erorilor, determinarea cauzei producerii lor și efectuarea corecturilor necesare în vederea eliminării acestora.

În practică fac obiectul depanării patru mari categorii de erori, corespunzător celor patru faze în tratarea unui program și anume:

a. *Erori de compilare,* care sînt puse în evidență în procesul compilării prin *lista cu mesaje de erori.* În această listă pentru fiecare eroare se precizează linia sursă în care s-a detectat eroarea, gravitatea erorii și un test explicativ care ajută programatorul să afle cauza producerii erorii.

Erorile din această grupă sînt în general cauzate de utilizarea incorectă a instrucțiunilor limbajului sau, la unele limbaje, de descrierea incorectă în formularul de programare (vezi îndeosebi limbajul COBOL). Aceste erori sînt numite și *erori sintactice.*

Deși limbajele de comandă și control acceptă trecerea în faza următoare (generarea modulelor obiect și editarea legăturilor) și în condițiile în care au fost detectate erori „mai puțin grave", optăm pentru a se elimina toate erorile, indiferent de nivelul de graviditate. Prin urmare, se impune o specificare expresă a parametrilor de compilare; cea mai mare parte a sistemelor de operare prezintă opțiuni implicite pentru parametri privind nivelul maxim de eroare care interzice sistemului trecerea la faza următoare compilării.

Tehnica eliminării erorilor care apar în faza de compilare este următoarea: se analizează mesajul din lista de erori; se identifică pe baza numărului

liniei sursă partea din progrm în care se găsește eroarea; se face corecția liniei sursă; se reia compilarea după corectarea tuturor erorilor.

b. *Erori de editare a legăturilor.* Modulele obiect obținute în faza de compilare sînt supuse în continuare fazei de editare a legăturilor în vederea obținerii programului în imagine memorie translatabilă (cu sau fără includerea unor module specializate în realizarea anumitor funcții cum ar fi module SGF*), modulele de bibliotecă etc.).

Erorile care apar în această fază sînt în general cauzate de: nerespectarea sintaxei comenzilor ce se adresează editorului de legături; definirea eronată a relațiilor între module și gruparea eronată a modulelor; incompatibilitatea între descrierea fișierului și parametrii furnizați referitori la fișierele memorate pe suporturi magnetice (vezi sistemele de operare pe FELIX C).

c. *Erori semnalate la încărcarea programelor.* Încărcarea unui program în memorie în vederea executării poate duce de asemenea la apariția unor erori care țin îndeosebi de dimensiunile programului de încărcat.

d. *Erori semnalate la executarea programelor.* În decursul executării unui program sistemul de operare emite mesaje care apar la consola operator (începutul unei lucrări, începutul unei faze, cerințele de resurse ale fazei etc.) și mesaje de eroare scrise atît la consola operator cît și la fișierul standard de ieșire (imprimanta) prin care se semnalează sfîrșitul anormal al unei lucrări.

Printre cauzele care duc la sfîrșitul anormal al unei lucrări cităm: apariția unei erori de intrare/ieșire ireparabilă, intervenția operatorului ca urmare a unui incident la unul din perifericele afectate programului, apariția unei erori de sistem, întîlnirea în timpul execuției a unui anumit tip de operație, trimiterea la o adresă inexistentă în memorie.

Depanarea programelor așa cum am menționat în primul paragraf a acestui capitol are ca obiectiv eliminarea erorilor ce apar într-un program pe parcursul tratării fazelor acestuia.

Este important să se pună la dispoziția programatorului instrumente care să permită depanarea de programe la un cost cît mai redus. Aceste instrumente de depanare se referă la comunicarea om-mașină: pe de o parte, programatorul trebuie să fie capabil să precizeze ce aspecte ale programului dorește să examineze și de ce informații are nevoie pentru localizarea erorilor, iar pe de altă parte, calculatorul trebuie să răspundă clar cu mesaje de diagnostic precise.

8.5.2. Tehnici de depanare

Printre tehnicile clasice și instrumentele folosite în depanare menționăm:

a) *Vidajul post-operativ,* în care sînt incluse informații cu privire la conținutul memoriei ocupate de program (obținut la cerere sau imprimat în mod obligatoriu în cazul despistării unei erori). Faptul că vidajul este exprimat în hexazecimal creează dificultăți în înțelegerea de către programatorii care lucrează în limbaje evoluate; în plus programatorul, în general, nu are posibilitatea controlării cantității de informații furnizate.

Pentru a fi utilizată în practică cu rezultate bune tehnica vidajului post-operativ trebuie să dea posibilitatea programatorului să comunice ce zonă din program dorește să fie listată și în ce format. Desigur că nu ar fi lipsit de interes să se prezinte programatorului informațiile în termenii programului sursă.

*) SGF — Sistem de Gestiune a Fișierelor.

Spre exemplu, în limbajul ALGOL W vidajul post-operativ furnizează informații programatorului cu privire la textul sursă, locul unde s-a produs eroarea, de asemenea se listează variabilele locale ale fiecărei proceduri în momentul erorii, iar variabilele care nu au fost inițializate se marchează.

b) *Vidajul-dinamic* apare în timpul execuției programului (fără să se oprească execuția) și este cerut prin instrucțiuni speciale de către programator. Programatorul specifică informațiile pe care le dorește (zona de vidaj, formatul etc.), iar cînd programul se execută și controlul ajunge la o astfel de instrucțiune are loc un salt la o rutină de vidaj care listează informațiile cerute, după care se continuă execuția în secvență a programului. În acest fel se poate urmări evoluția dinamică a variabilelor, tablourilor etc. Desigur că pentru a nu încărca prea mult programul cu astfel de instrucțiuni ele trebuie plasate numai în locurile cheie ale programului. Această tehnică este des folosită de utilizatorii sistemului MIX al minicalculatorului I-100. Programatorii care utilizează limbajul MACRO au posibilitatea să insereze în programul lor macroinstrucțiuni speciale: SNPBK$ pentru definirea formatului de listare (ASCII, octal, RADIX-50) și a zonei de vidare și SNAP$ pentru provocarea vidajului. În alte sisteme (ca de exemplu Felix C) vidajul are loc numai în cazul terminării programului din cauza unei erori sau la cerere la sfîrșitul execuției normale; el reprezintă starea programului în momentul opririi execuției.

c) *Trasarea*, care permite în momentul execuției, furnizarea unor informații în legătură cu fiecare instrucțiune. Aceasta duce implicit la o scădere importantă a vitezei de execuție fapt pentru care trasarea trebuie aplicată numai la porțiuni de program și îndeosebi în faza de punere la punct a programului pentru erori greu de detectat.

d) *Documentarea*, ca tehnică folosită în depanarea programelor vizează deopotrivă: textul programului sursă, tabelele de simboluri pentru module, indexul de variabile, schema logică de sistem și de detaliu pe module și manualul de utilizare al programului.

Odată cu apariția sistemelor care funcționează după principiile timpului divizat și în regim conversațional s-a îmbunătățit și modalitatea de depanare a programelor: programatorul, care lucrează în conversațional cu calculatorul are posibilitatea să acționeze direct asupra programului său. El are posibilitatea de acces, în regim interactiv, la diferite mecanisme de control prin care poate modifica comportarea programului aflat în faza de testare.

Sistemele care permit depanarea interactivă trebuie să asigure:

— comenzi pentru introducerea și eliminarea punctelor de oprire (breakpoints) în program. Dacă în momentul execuției se ajunge la un punct de oprire programul este întrerupt, starea se „îngheață" și controlul trece programatorului la terminal;

— posibilitatea sistemului de a reda controlul programatorului în cazul în care se detectează o eroare în program (de exemplu, împărțirea prin zero, extragerea rădăcinii pătrate dintr-un număr negativ etc.); cînd apare eroarea se imprimă un mesaj la terminal după care se dă controlul programatorului pentru noi investigații;

— posibilitatea de a examina și modifica orice locație a programului din moment ce starea lui a fost „înghețată" și programatorul a primit controlul la terminal;

— posibilitatea de a relansa programul, nu neapărat de la punctul în care a fost întreruptă execuția;

— posibilitatea de editare de text.

8.5.3. Facilitatea testării și depanării

Operațiile de testare și depanare sînt influențate de următorii factori:

A) *Documentarea* are un rol fundamental în efectuarea operațiilor de testare și depanare și în acest sens o atenție deosebită trebuie acordată autodocumentării programelor și respectării unor reguli de stil în realizare. Este indicat ca fiecare program să fie însoțit de un dicționar de identificare în care să apară numele identificatorului și locurile din program în care este definit și respectiv utilizat. Sînt utile și informațiile cu privire la condițiile și expresiile condiționale utilizate în program.

B) *Folosirea în cadrul programelor a unor rutine de trasare sau instrucțiuni de punere la punct,* care permit tipărirea anumitor mesaje sau valoarea anumitor variabile pentru a scoate în evidență operațiile din momentul execuției, memorarea și tipărirea unor informații prin completarea celor furnizate de vidajul memoriei etc. Cea mai simplă acțiune pe care programatorul o poate executa este includerea unui număr de instrucțiuni de tipărire care îi permit să urmărească execuția programului. Acestea pot imprima rezultate intermediare, valori de variabile sau chiar mesaje pentru a indica dacă o subrutină anume a fost apelată sau nu. Comentariile introduse în programe în punctele critice pot fi însoțite de instrucțiuni speciale care verifică dacă condițiile sînt îndeplinite (spre exemplu dacă un parametru al unei subrutine aparține unui interval).

În mod normal codul scris în scopul ușurării depanării trebuie înlăturat odată ce programul a fost testat. Pentru a servi și în fazele ulterioare de întreținere a unui program codul de depanare poate fi trecut în categoria comentariilor (folosind anumiți parametri din structura comenzilor adresate sistemului de operare). Exemple de instrucțiuni care fac parte din codul de depanare:

— instrucțiunea DEBUG (cu opțiunile UNIT, SUBCHK, TRACE, INIT, SUBTRACE) în limbajul FORTRAN, care prin opțiunile sale definește condițiile de execuție a operațiilor de punere la punct și indică pe acelea aplicabile la program în întregime;

— instrucțiunea PRINT în limbajul COBOL cu structura

```
Col.  Col.
 7    12        {nume-dată-1}  [{nume-dată-2}]...
 +    PRINT     {literal-1   }  [{literal-2  }]
```

care permite urmărirea dinamică a programului (traseul parcurs în procesul de prelucrare) precum și a unor clauze de control cum ar fi depășirea de capacitate (ON SIZE ERROR) etc;

— în limbajul dBASE în faza de punere la punct a programelor prin comanda SET TALK ON rezultatele executării comenzilor sînt afișate pe ecran.

C) *Modularizarea programelor* oferă avantaje evidente încă din faza de concepere a programelor. Descompunerea pe module asigură punerea în evidență a elementelor de interfață ce permite folosirea unor tehnici evoluate de testare. Testarea trebuie realizată astfel încît să permită verificarea simultană și a elementelor care integrează un modul în cadrul programului și a elementelor care particularizează modulul în cadrul programului.

D) *Alegerea strategiei de testare* care să asigure într-un concept top-down *testarea funcțională* pentru a verifica dacă programul furnizează rezultatele cerute de utilizator, *testarea logică* care verifică modul în care sînt efectuate calculele și *testarea interfețelor* care are ca obiectiv verificarea corectitudinii

transmiterii de date, de legături între unitățile de program, a tipului și a concordanței parametrilor.

E) *Parametrizarea* oferă avantaje mai ales pentru adaptarea, modificarea și dezvoltarea programului. De asemenea, în faza de depanare cînd se cer anumite schimbări în program parametrizarea se dovedește a fi foarte utilă.

Practica dovedește că este foarte dificil să se modifice o secvență de calcul în care un parametru nu este declarat explicit. În general toate limbajele de înalt nivel permit declararea de parametri, iar utilizarea acestei facilități ajută pe programatori să îmbunătățească mult calitatea produselor.

F) *Includerea în program a unor subrutine sau instrucțiuni* care permit obținerea de informații în legătură cu comportarea programului în timpul execuției: de cîte ori este executată o instrucțiune, lista tuturor instrucțiunilor sau procedurilor executate, lista variabilelor executate sau repetate etc. Aceste date pot fi utile și la evaluarea performanțelor.

G) *Descrierea datelor de test* constituie un factor deosebit de important dacă avem în vedere că un program se poate reprezenta ca o funcție de forma:

Rezultate = f(Date)

Alegerea datelor de testare trebuie astfel făcută încît prin mulțimea datelor de intrare stabilite să se dea posibilitatea testării, dacă este posibil, a tuturor instrucțiunilor și prin urmare a tuturor drumurilor din program. În cazul în care numărul combinărilor posibile este mare, studiul în scopul alegerii datelor de test va urmări punctele relevante (nodale) ale programului. Dacă programul a fost bine modularizat și structurat alegerea datelor de testare este mult ușurată. Datele de test selectate trebuie să asigure atît testarea funcțiilor programului, cît și testarea punctelor slabe ale lui (depășirea capacității fișierelor, tabelelor etc.) precum și a cazurilor de catastrofă (cazurile în care o eroare ar produce pagube mari). În vederea unei mai bune generări a datelor de test se pot folosi produse-program de generare automată a acestora (vezi produsul program GENDATA existent în BNP).

8.6. Integrarea programelor și testarea de ansamblu

Conceperea și dezvoltarea după metode moderne a programelor a însemnat trecerea de la programarea individuală artizanală la programarea colectivă. În acest nou context performanțele mari se obțin prin asamblări de module (subprograme/proceduri/rutine).

Un modul realizează o funcție de sine stătătoare și este pus în relație cu celelalte module.

Aceasta permite proiectarea și dezvoltarea independentă a fiecărui modul precum și testarea lui pentru a vedea dacă realizează funcția prestabilită. Asupra fiecărui modul se realizează un control de calitate și se fac eventuale propuneri în vederea optimizării codului.

Modulele astfel *testate* fac în continuare obiectul unui *proces de integrare*, adică de asamblare în cadrul sistemului.

Odată cu integrarea componentelor se poate trece la testarea de ansamblu adică a produsului program pentru a verifica corectitudinea realizării funcțiilor integrate.

Se verifică astfel dacă fiecare componentă furnizează informațiile necesare celorlalte componente și dacă funcția de ansamblu a produsului program este realizată conform cerințelor sistem elaborate de beneficiar.

Organizarea procesului de integrare este în funcție de strategia adoptată pentru elaborarea produsului-program.

Astfel, în varianta adoptării strategiei ascendente procesul de integrare începe după ce au fost elaborate modulele de pe nivelul cel mai de jos al ierarhiei.

Această strategie de integrare prezintă dezavantajul că impune construirea unor componente care pe măsură ce sînt elaborate cele de nivel ierarhic superior sînt eliminate din sistem (este vorba în special de module care asigură interfața pînă cînd un anumit modul este realizat).

Dacă se adoptă strategia dezvoltării descendente a programelor integrarea începe cu modulele de pe nivelele ierarhice din partea de sus ale ierarhiei. În condițiile folosirii metodelor moderne de proiectare și programare se recomandă adoptarea strategiei integării descendente.

Această strategie prezintă avantaje în desfășurarea procesului integrării. Un prim avantaj constă în faptul că nu mai face necesară construirea unor module care în final nu vor fi incluse în produsul-program. Un al doilea avantaj constă în faptul că integrarea unor module se poate desfășura în paralele cu proiectarea și codificarea altor module.

Reușita în integrarea programelor presupune tehnici adecvate, disciplină în muncă și strategie corespunzătoare.

Realizarea integrării revine îndeosebi programatorului șef care coordonează întreaga activitate de elaborare a produsului-program.

Testarea și depanarea constituie un proces complex dar important în ciclul de elaborare al programelor. Așa cum se afirmă în [3, 46, 88], costul testării unei modificări într-un sistem de mari proporții poate fi de 100 de ori mai mare decît costul realizării acelei modificări. Din această cauză testarea și depanarea programelor au stat în permanență în atenția cercetătorilor. Am enunțat în cadrul acestui capitol tehnici, strategii și instrumentele folosite în testare, care presupun abordarea într-o manieră mai eficientă a activității de testare. O tratare exhaustivă este în general aproape imposibilă. De un folos real se bucură programele de testare automată care asigură în mod considerabil creșterea productivității în programare.

Literatura de specialitate [60] citează programe pentru analiza statică, analiza dinamică și testarea dinamică asistată de calculator.

Programul care realizează analiza statică a programelor are funcții înrudite cu cele ale compilatoarelor fapt pentru care sînt păreri că ar fi mai bine ca ele să fie incluse în compilatoare.

Programele care realizează analiza dinamică sînt mult mai complexe ele urmînd să furnizeze informații cu privire la: șirurile de valori atribuite variabilelor; secvența instrucțiunilor executate; influența unor valori anterior calculate asupra valorii curente a unei anumite variabile; efectele erorilor din datele programului asupra preciziei rezultatelor; statistici asupra utilizării resurselor, etc.

Procedurile de testare descrise într-un limbaj specific pot fi analizate de un program de testare dinamică asistat de calculator.

Progresele realizate în plan metodologic au produs și unele modificări în organizarea activității de testare a programelor. Considerînd că în elaborarea programelor folosim o ierarhie de limbaje, iar pe fiecare nivel din ierarhie rezultatul activității desfășurate este un program (care în general numai pe ultimul nivel constituie un program acceptabil calculatorului) pe fiecare nivel ierarhic trebuie să se asigure și testarea programului obținut. Va exista spre exemplu elaborarea proiectului programului și respectiv testarea proiectului obținut. În felul acesta erorile detectate se pot analiza și elimina cu un cost cît mai redus.

Extinderea folosirii în practică a metodelor de verificare a corectitudinii programelor va duce implicit la reducerea considerabilă a costului testării.

9. Documentarea programelor

9.1. Cerințele documentării programelor

Trecerea de la artizanat la industrie în activitatea de programare a însemnat cerințe noi și pe linia documentării programelor.

Programele generalizabile, folosite de un număr tot mai mare de utilizatori trebuie să fie însoțite de o documentație clară și completă care să conțină informații în legătură cu funcția programului, structura generală și de detaliu a acestuia, datele de intrare și ieșire, specificațiile de realizare, implementare și exploatare.

Se apreciază că documentarea programelor deține peste 10% din fondul de timp afectat activității de programare. În vederea realizării unui fond național de produse program pe lîngă faptul că s-au intensificat preocupările pe linia creșterii performanțelor în programare s-au stabilit și condițiile pe care trebuie să le îndeplinească un astfel de produs-program.

Astfel Normativul elaborat de Institutul Central pentru Conducere și Informatică [62]*) precizează că pentru a fi inclus în fondul național de produse-program elaboratorul trebuie să prezinte documentația acestuia întocmită conform indicațiilor metodologice, și o formă de prezentare corespunzătoare normelor cadru BNP**).

Documentația întocmită, odată cu elaborarea programului trebuie să se răspundă următoarelor cerințe [63]:

a. să conțină informații în legătură cu organizarea generală a produsului-program într-o concepție tehnico-comercială;

b. să conțină informații în legătură cu domeniul de aplicare, descrierea intrărilor, descrierea ieșirilor, organizarea datelor, descrierea procedurilor automate și estimarea performanțelor;

c. să conțină informații în legătură cu descrierea resurselor necesare, pregătirea programului pentru rulare și exploatare, datele de intrare, pregătirea sistemului pentru rulare, datele de ieșire.

d. să conțină informații în legătură cu organizarea logică și tehnică a programului astfel încît să permită efectuarea unei intervenții și de un alt programator decît cel care l-a elaborat în vederea realizării întreținerii.

În accepțiunea generală, așa cum am menționat, produsul-program înglobează programele înregistrate pe suport și documentația finală a acestora.

În structura acestor cerințe, delimităm documentația de prezentare a produsului-program, documentația de utilizare; documentația de exploatare și documentația de realizare.

---

*) În referințele bibliografice s-au păstrat denumirile institutelor anterioare înființării Institutului de cercetare științifică și inginerie tehnologică pentru tehnică de calcul și informatică.

**) BNP—Biblioteca Națională de Programe.

În tabelul 9.1 prezentăm sintetic tipurile de documentații și destinația acestora.

Tabelul 9.1. Tipuri de documentație pentru un produs-program

Nr. crt.	Tip documentație	Conținutul	Destinat
1.	Documentație de prezentare	Informații generale de natură tehnică și economică ale produsului-program	Utilizatorilor potențiali (organe de conducere)
2.	Documentație de utilizare	Informații privind utilizarea curentă a produsului-program (interfețe proceduri manuale-proceduri automate)	Utilizatorilor efectivi care își desfășoară activitatea în domeniul de aplicație al produsului-program
3.	Documentație de exploatare	Informații privind ordonarea lucrărilor (job-urilor) și rularea pe calculator	Utilizatorilor operatori (exploatare produs-program)
4.	Documentație de realizare	Informații cu privire la componentele produsului-program; proiectul logic, proiectului tehnic programele, datele de test etc.	Specialiști în întreținere și dezvoltare

Evoluția spre informatica distribuită și sisteme de calcul în lucru în regim interactiv produce unele mutații în elaborarea documentației produselor-program îndeosebi în ceea ce privește documentația de utilizare și cea de exploatare.

De asemenea, folosirea unor instrumente de ajutor în realizarea activității de documentare a produselor-program va avea implicații în organizarea și realizarea documentării.

Prin analogie se poate afirma că documentația programelor reprezintă „cartea tehnică a acestora"; numai în baza ei se poate realiza folosirea programelor în condiții de eficiență.

Numai cel ce nu a condus niciodată un proiect de elaborare sau întreținere a unui produs-program nu știe ce înseamnă să caute o referință cu privire la un parametru sau rolul și contextul de utilizare al unei proceduri. Ca să nu mai vorbim de programe abandonate întrucît nu există o documentație corespunzătoare în vederea întreținerii și adaptării unor noi condiții.

Tendința care se manifestă la unii specialiști de a ajunge cît mai repede la program, urmînd ca în final să constituie documentația corespunzătoare nu o considerăm indicată. Dacă în urma testării și validării produsului-program se definitivează și documentația, întocmirea documentației începe odată cu elaborarea produsului-program.

Desigur că trebuie făcută distincție între documentația de lucru folosită de membrii echipei de programare pentru a comunica între ei, documentația de avizare, folosită în vederea controlului și documentării parțiale și documentația finală care însoțește produsul-program.

În continuare vom trata numai problemele cu privire la documentația finală.

În încheierea acestui paragraf menționăm că în procesul organizării documentației se iau decizii cu privire la: informația ce trebuie inclusă în documentație; cum trebuie să se organizeze aceste informații; cum trebuie să se prezinte documentația pentru a putea fi înțeleasă cît mai ușor de utilizator.

Este important prin urmare să se știe ce să se includă în documentație și pînă la ce nivel să se coboare (în fiecare document) în prezentarea de detaliu a fiecărui subiect.

Organizarea se referă la structurarea materialului; se rezolvă astfel problemele legate de decupare și grupare într-o anumită ordine pentru a asigura accesibilitate maximă. Accesibilitatea depinde de felul în care subiectele conținutului devin vizibile prin antet, titluri de paragrafe ș.a.

În privința organizării un rol deosebit îl au: tablele de materii; indexul termenilor speciali; subtitlurile paragrafelor ce apar pe fiecare pagină; gruparea informației în paragrafe sau capitole; tipizarea formularelor folosite în documentație.

Formatul se referă la punerea în pagină, adică la felul de prezentare a informațiilor în text. Regula principală este că limbajul de prezentare a informației trebuie să fie legat de prezentarea figurilor din text.

Figurile trebuie să joace un rol deosebit în prezentarea conținutului. Descrierea subiectului pe pagina din stînga cu prezentarea figurii pe pagina din dreapta contribuie în mod deosebit la facilitarea înțelegerii.

9.2. Tipuri de documentații

În volumul 1 s-a precitat că în accepțiunea curentă un produs-program cuprinde unul sau mai multe programe, care din punct de vedere al funcției îndeplinite creează o unitate logică, documentația finală și suportul de memorare a acestora.

În funcție de destinația lor delimităm două tipuri de documentații (pentru fiecare produs-program) și anume:

— *documentații destinate utilizatorului* în care includem (în principiu): documentația de prezentare, documentația de utilizare și documentația de operare;

— *documentații destinate echipei de întreținere* a produsului-program în care includem documentația de realizare.

Documentația de prezentare este destinată în special organelor de conducere ale utilizatorului (real sau potențial) și are un pronunțat caracter tehnico-comercial. În cadrul acestei documentații se cuprind informații de ordin general cu privire la domeniul de aplicație și posibilitățile oferite de produsul-program.

Tot în această documentație se prezintă schema funcțională a produsului program, intrările și ieșirile (dacă este posibil cu modele concrete), varianta de rezolvare a problemei, organizarea și gestionarea datelor, algoritmii folosiți în prelucrare, resursele necesare, performanțe obținute în faza de testare/experimentare și eventual posibilitățile de cuplare/integrare cu alte produse-program.

Pe baza acestei documentații utilizatorul realizează o primă cunoaștere a produsului-program, a posibilităților pe care acesta le oferă și eventual

poate face alegerea între mai multe produse program realizate pentru rezolvarea aceleiași probleme.

Documentația de utilizare conține toate informațiile necesare utilizatorului pentru a asigura folosirea, în condiții de funcționare normală a produsului-program.

Principalele capitole ale manualului de utilizare în structura indicațiilor metodologice actuale sînt [63, 103]:

a) *Descrierea domeniilor de aplicare:* se prezintă aria de probleme acoperită cu precizarea funcțiilor/procedurilor executate automat, principalele tipuri de beneficiari; variante de aplicare (limite, restricții), prezentarea generală a metodelor, modulelor, algoritmilor utilizați, structura generală a produsului-program, resursele necesare pentru exploatarea produsului program, facilități de generare și adaptare a produsului-program.

b) *Descrierea intrărilor:* se precizează tipurile de intrări și modurile de colectare a datelor de intrare (lista intrărilor, specificarea surselor generatoare de date primare, frecvența de apariție, documente primare) și procedurile de codificare, validare și transmitere a datelor de intrare.

c) *Descrierea ieșirilor:* se precizează tipurile de ieșiri și modurile de obținere a datelor de ieșire (lista ieșirilor, specificarea destinațiilor ieșirilor, rapoarte de ieșire), procedurile de interpretare a ieșirilor etc.

d) *Descrierea organizării datelor:* se prezintă soluția adoptată pentru organizarea și gestionarea datelor și descrierea fișierului/bazei de date.

e) *Descrierea funcțională a procedurilor automate:* se precizează tipul de calculator și sistemul de operare, schema funcțională generală (în care se indică și principalii pași ai procesului de rulare și graficul de exploatare pe calculator, descrierea fiecărei proceduri și a modului de lansare în execuție, restricții în aplicarea procedurilor, erori și proceduri de tratare a erorilor, mesajele emise și acțiunile de intreprins, specificarea cazurilor de terminare anormală a programelor, reluarea programelor la întrerupere, condiții de reluare, situațiile în care pot apare mesaje de eroare emise de sistem.

f) *Estimarea performanțelor:* se precizează capacitățile de memorie externă necesară în raport cu dimensiunea fișierelor/colecțiilor de date, estimarea timpilor de execuție în funcție de volumul datelor de intrare, aprecierea performanțelor de timp ale produsului-program.

g) *Exemple.*

Metodele actuale prevăd, pe capitole, înregistrarea informațiilor în formulare tipizate.

Este indicat ca la întocmirea documentației de utilizare să se folosească combinat tehnica textelor cu cea a formularelor tipizate.

Indicațiile metodologice actuale privind realizarea sistemelor informatice și a produselor-program prevăd folosirea unor formulare tipizate întocmite la nivel de produs-program iar unele dintre ele și la nivelul componentelor principale ale acestuia.

Dosarul de utilizare constituie și suportul de curs în vederea formării personalului pentru implementarea unui produs-program.

Documentația de exploatare conține informații cu privire la rularea pe calculator a produsului-program.

Informațiile din documentația de exploatare sînt structurate pe următoarele capitole [63, 103]:

a) *Descrierea resurselor necesare:* se definește configurația de echipamente și dispozitive auxiliare necesare pentru rularea fiecărei lucrări/proceduri

distincte din cadrul produsului-program și sistemul de operare sub care produsul este operațional.

b) *Pregătirea produsului pentru exploatare* — se prezintă proceduri de aducere a produsului din forma livrabilă în formatul exploatabil necesar utilizatorului, organizarea bibliotecii, condițiile necesare pentru exploatarea produsului-program, configurația, programele utilitare necesare să figureze în biblioteca utilizatorului.

c) *Datele de intrare* — se prezintă lista fișierelor cu date de intrare (suporți, condiții de validare etc.), descrierea procedurilor de control și verificare a datelor de intrare prealabile rulării, descrierea datelor de intrare folosite în exemplele de test.

d) *Pregătirea sistemului pentru rulare:* se descriu poziționările inițiale a tuturor contactelor și cheilor de la consolă și de la celelalte dispozitive ale sistemului implicate în rularea produsului-program, enunțurile de control și comenzile operatorului necesare pentru pregătirea sistemului înainte de execuție etc.

e) *Procedurile de exploatare a produsului* — se descriu programele de control ce se folosesc pentru diferite tipuri de înlănțuiri, programele de control pentru aplicațiile standard, alocări și realocări de memorie, dimensionarea volumelor pentru fișiere, indicații privind scrierea cartelelor legate de fișiere, procedurile de exploatare referitoare la toate fazele de execuție (crearea fișierului/bazei de date, actualizarea fișierului/bazei de date, exploatarea produsului-program, procedurile de salvare a datelor, constituirea versiunilor și generațiilor de fișiere, procedurilor de refacere a lucrărilor pornind de la generațiile de date anterioare, proceduri catalogate.

f) *Terminarea lucrării* — se prezintă specificațiile privind: criteriile de apreciere a terminării normale a lucrării, specificațiile cazurilor de terminare anormală a lucrării și a intervențiilor (acțiunilor operatorului pentru fiecare caz).

g) *Datele de ieșire:* se descriu tipurile și categoriile de date de ieșire obținute în cazul rulărilor normale.

h) *Instrucțiuni specifice utilizării echipamentelor de colectare a datelor:* se fac precizări cu privire la utilizarea acestei categorii de echipamente.

Indicațiile metodologice actuale prevăd și în cazul acestei documentații combinarea tehnicii textelor cu cea a formularelor tipizate.

În această documentație indicațiile metodologice prevăd și listinguri cu privire la procedura de aducere a produsului program din formatul primit de la elaborator sau dintr-o bibliotecă de programe în formatul exploatabil necesar utilizatorului precum și organizarea bibliotecii de programe, descrierea procedurilor de control și verificare a datelor de intrare prealabile rulării; descrierea datelor de intrare folosite în exemplele de test, programe de control ce se folosesc pentru diferitele tipuri de înlănțuiri, procedurile de exploatare referitoare la toate fazele de execuție și alte indicații legate de exploatarea curentă a produsului-program.

Dosarul de exploatare constituie și suportul de curs în vederea formării personalului desemnat pentru implementarea și exploatarea produsului-program.

Documentația de realizare conține informații cu privire la proiectul logic și tehnic al produsului-program și produsul-program în format sursă. Ea este destinată îndeosebi echipei de întreținere în vederea asigurării funcționării normale a produsului-program și actualizării acestuia.

Informațiile sînt structurate pe capitole astfel:

— proiectul logic și tehnic de detaliu al produsului-program;

— documentația programelor/modulelor program: schema bloc, arborele de structură, funcțiuni, variante funcționale, descriere, schema logică, textul sursă autodocumentat, datele utilizate organizate pe suporți externi și în memoria internă, legături cu celelalte module, mesaje, acțiuni, listinguri martor (programe și proceduri de testare, proceduri de exploatare, proceduri catalogate, conținut biblioteci), indicații privind întreținerea produsului-program.

Și pentru întocmirea acestei documentații se indică folosirea formularelor tipizate și a listingurilor.

Ea trebuie astfel organizată încît să permită accesul cu ușurință la informații în vederea întreținerii, adaptării și dezvoltării produsului-program.

O importanță deosebită se acordă listingurilor martor autodocumentate cu privire la textul sursă; proceduri de exploatare, proceduri catalogate, conținut biblioteci etc.

9.3. Instrumente de documentare

În condițiile în care elaborarea și întreținerea produselor-program antrenează un număr însemnat de persoane — deopotrivă, utilizator și realizator —, iar documentarea programelor devine o cerință primordială pe linia reducerii costurilor, în domeniul informaticii a apărut ca necesară dezvoltarea unor instrumente de ajutor în programare cu funcții pe linia documentării automate a produselor-program.

Marea majoritate a sistemelor de proiectare și generare automată a programelor existente în practică au printre obiective și elaborarea automată a documentației în structura prezentată în sinteză în paragraful precedent.

Experiența practică demonstrează că elaborarea documentației unor proiecte mari și actualizarea acesteia este foarte dificil de realizat, fapt pentru care se justifică crearea unei baze de date documentare gestionate automat și dacă este posibil, punerea în relație directă a acesteia cu baza de programe pe care o documentează.

Adesea se constată că modificarea unei documentații în manuscris nu este simplă, trebuie adesea să se recopieze mult mai multă documentație decît informația care este modificată sau adăugată.

Aceste dificultăți determină pe programatori la întocmirea unei documentații a posteriorii (după dezvoltarea programelor) care de cele mai multe ori este incompletă și chiar inexactă.

Dacă însă documentația este ținută la zi, într-o *bază de date documentară* (pe un suport magnetic), sînt introduse cu ușurință modificările (folosind la limită un editor de texte) și prin proceduri adecvate, efectuată punerea la zi a întregii documentații.

Orice utilizator poate obține ultima versiune a documentației.

În varianta elaborării specificațiilor sub formă de text liber în limbaj natural se poate folosi ca instrument software un editor de texte.

În sistemul de operare pentru minicalculatoarele INDEPENDENT și CORAL se utilizează programe speciale pentru elaborarea documentației (RUNOFF, TEC etc.) care rezolvă problema punerii în pagină, numerotarea paragrafelor, capitolelor, plasarea figurilor în pagină etc.

Folosirea în practică a unor produse program cu rol de instrument de documentare permite: formalizarea și structurarea specificațiilor de definire a problemelor, verificarea coerenței, corectitudinii și completitudinii documentației, structurarea unitară a documentației, crearea unei baze unice de date folosită deopotrivă de utilizator, proiectant și cel în sarcina căruia cade întreținerea produsului-program. Totodată s-a permis automatizarea în mai mare parte a relației *documentație-program* spe toate etapele ciclului de viață al programelor.

Procedurile de gestionare automată a documentației — incluse în sistemul programelor de ajutor în programare — permit utilizarea *bazei de date pe niveluri ierarhice*, unde unui nivel ierarhic îi corespunde un tip de documentație.

Realizările practice în acest domeniu, cum ar fi proiectul ISDOS dezvoltat la Universitatea din Michigan — S.U.A. și sistemul PACSIN — dezvoltat în cadrul ICI au ca obiective:

a) punerea la dispoziția utilizatorului a unui limbaj de elaborare a specificațiilor de descriere a produsului-program;

b) automatizarea activității de analiză a specificațiilor în vederea verificării coerenței, corectitudinii și completitudinii;

c) automatizarea activității de proiectare și dezvoltare a programelor;

d) gestionarea automată a documentației.

Pe microcalculatoare cu lucru sub controlul sistemului de operare CP/M în elaborarea documentației se poate folosi cu rezultate foarte bune editorul de texte WS.

9.4. Autodocumentarea programelor și creșterea accesibilității

Pentru ca programele să fie înțelese și întreținute de către un programator — altul decît cel care le-a elaborat — trebuie să conțină, în cadrul liniilor sursă suficiente explicații cu privire la funcția îndeplinită, datele de intrare, parametrii de lucru, procedurile de activat și eventual semnificația codificărilor folosite.

Un program care conține astfel de informații spunem că este un program autodocumentat.

Din experiența autorilor rezultă că autodocumentarea constituie un factor deosebit de important pe linia reducerii costului întreținerii programelor.

Pe linia autodocumentării programelor se înscriu și cerințele cu privire la folosirea unor identificări sugestive pentru a denumi fișiere, cîmpuri de date, entități, parametri, proceduri, paragrafe etc.

Un program autodocumentat va conține *suficiente* și în același timp *concise* informații cu privire la datele de supus prelucrării și alogoritmul de prelucrare. Ele sînt ușor accesibile dacă la redactarea lor s-au respectat regulile de aliniere și scriere.

În principiu se indică scrierea pe o linie distinctă a fiecărei instrucțiuni în limbaj sursă.

Programele devin ușor accesibile dacă la scrierea lor s-au respectat regulile de aliniere corespunzător apartenenței la un anumit grup ierarhic.

Spre exemplu, liniile sursă (în COBOL) ce urmează se consideră corect aliniate ierarhic:

```
ANALIZA TEXT
* Procedură ce analizează un caracter al liniei introduse
* de la terminal (sau din cartelă) și inserează caracterele
* de trecere a terminalului în mod de editare
* video invers și de revenire în modul normal de editare, după caz
* Fiecare caracter din linia sursă introdusă de
* la terminal este analizat
* Caracterul „<" din cadrul liniei sursă precizează trecerea termina-
* lului în regim video invers ceea ce înseamnă inserarea caracterelor co-
* respunzătoare desemnate prin RZ11 și RZ12.
* Caracterul „>" din cadrul liniei sursă precizează revenirea termi-
* nalului în regim normal de editare și prin urmare inserarea caractere-
* lor desemnate prin RZ21 și RZ22.
* Linia este declarată sub formă de tablou
* sub numele CARACTER. Variabilele V și respectiv F constituie
* indicii de adresare a valorilor tablourilor care desemnează
* linia introdusă de la terminal CARACTER și respectiv linia
* obținută în urma inserării caracterelor de editare în regim special
CARACTERD
ADD     1     To  F
IF  CARACTER (V) „<"
THEN
        MOVE RZ11 TO CARACTERD (F)
        ADD  1    TO F
        MOVE RZ12 TO CARACTERD (F)
ELSE
        IF     CARACTER (V)     „>"
               THEN
                    MOVE RZ21 TO CARACTERD (F)
                    ADD  1    TO F
                    MOVE RZ22 TO CARACTERD (F)
               ELSE
                    MOVE CARACTER (V) TO CARACTERD (F)

* ENDIF
* ENDIF
* SF—ANALIZA-TEXT.
EXIT
```

După cum se poate observa comentariile dețin un loc important în cadrul programului. Tocmai pentru a scoate mai bine în relief rolul comentariilor nu s-a mai urmărit conciziunea acestora.

Autodocumentarea programelor vizează deopotrivă și aspectele de utilizare a acestora. Pentru produse-program utilizate în mod interactiv se consideră deosebit de util facilitățile ca prin comenzi „HELP" să se obțină informații asupra posibilităților de lucru la nivel produs-program.

Pentru aceasta trebuie ca la elaborarea produsului-program să se creeze fișierul cu textele de „HELP" sau să se includă în program texte de „HELP".

Comenzile „HELP" se dovedesc deosebit de utile în dezvoltarea interactivă a programelor și cu precădere în generarea de cod folosind un limbaj de programare evoluat.

Revenind la exemplul prezentat anterior un prim text „HELP" poate preciza funcția procedurii ANALIZA-TEXT. Prin alte texte „HELP" se pot obține informații cu privire la semnificația și eventual valorile datelor folosite în cadrul procedurii.

Folosirea textelor tip „HELP" pentru a ușura considerabil munca utilizatorului produsului-program constituie, așa cum vom vedea în paragraful 15.2 o cale importantă pe linia creșterii calităților ergonomice ale programelor.

Totodată, alături de folosirea tehnicii „meniurilor" ele ușurează și în același timp *umanizează interfața om-calculator*. Problema creșterii accesibilității programelor a devenit cu atît mai importantă cu cît înregistrăm în ultimul timp o creștere considerabilă a numărului sistemelor de dezvoltare în regim interactiv a programelor (vezi sistemele UNIX, RSX, CP/M etc.).

9.5. Documentații pe tipuri de utilizatori

În paragrafele precedente au fost prezentate îndeosebi tipurile clasice de documentații (manuale). Realizarea unor produse-program complexe și diversificate pe domenii de aplicații impune o viziune nouă și în ceea ce privește documentațiile.

Pe lîngă aspectele legate de gestiunea bazelor de date asociate proiectului apar probleme pe linia construirii unor documentații în funcție de tipurile de utilizatori.

În [129] se precizează că deoarece un SGBD (Sistem de Gestiune a Bazelor de Date) colaborează cu mai mulți factori umani (programatori de aplicații, utilizatorul neinformatician, administratorul bazei de date), fiecare factor utilizînd într-un mod specific anumite funcțiuni ale SGBD-ului și documentația va trebui să fie adecvată acestor factori.

Întocmirea unor documentații pe tipuri de utilizatori se pune în mod asemănător și pentru alte tipuri de produse-program de complexitate asemănătoare.

Dezvoltarea sistemelor interactive impune, spre exemplu, documentații adecvate competenței asociate nivelului ierarhic al utilizatorului. Așa cum am mai menționat aceste documentații se pot obține, la cerere, automat făcînd apel la funcțiile sistemului de gestiune a bazei de date asociate proiectului. Ea poate fi și integrată în produsul program folosind tehnica HELP-urilor prin care se dau, gradat, explicații care să permită utilizatorului realizarea dialogului cu calculatorul în condiții optime.

10. Implementarea programelor

10.1. Considerații generale

Termenul de implementare cunoaște în literatura de specialitate mult mai multe accepțiuni.

Astfel prin *implementare a programului* înțelegem acea parte a activității de programare care are ca obiectiv codificarea, testarea modulelor și integrarea acestora într-un tot unitar numit *program*. Conform acestei accepțiuni activitatea de programare cuprinde ca principale componente: proiectarea programului; implementarea programului.

Cum aspectele legate de codificare, testare și integrare au fost tratate în capitole distincte în cele ce urmează vom prezenta implementarea în accepțiunea de activitate care are ca obiectiv adaptarea unor produse program de uz general la cerințele utilizatorilor și aducerea acestora în formă operațională la utilizator.

Așa cum s-a menționat în capitolul 1, preocupările pe tema utilizării resurselor din unitățile de informatică cu eficiență competitivă au impus proiectarea și realizarea unor programe pe tipuri de prelucrări, domeniu de activitate și grupe tipologice de unități economice.

Produsele-program astfel obținute împreună cu documentația corespunzătoare (manualul de utilizare, manualul de exploatare, manualul de realizare și manualul de prezentare) sînt organizate în biblioteci de programe (după ierarhia: bibliotecă la nivel centru de calcul, biblioteci teritoriale și biblioteca națională).

În aceste condiții utilizatorii au de ales între varianta proiectării și realizării propriului produs-program și cea a preluării și implementării unui produs program-existent într-una din bibliotecile mai sus menționate.

Deși cea de a doua variantă este mult mai avantajoasă, în realitate folosirea unor produse-program pe măsură este de regulă mai performantă în exploatare. Trebuie, de asemenea, precizat că eficiența implementării unui produs-program preluat dintr-o bibliotecă de programe ține de modul în care au fost respectate principiile și folosite tehnicile, procedeele și instrumentele de inginerie a programării.

Opțiunea unor utilizatori pentru proiectarea și realizarea unor produse-program pe măsură, deși poate fi mai costisitoare, menține avantajul psihologic generat de participarea beneficiarului pe parcursul elaborării, ceea ce determină un plus de încredere.

Experiența practică demonstrează că mai sînt unități economice care manifestă încă rețineri în implementarea de produse-program din biblioteci de programe pe considerentul că încercările făcute cu multe dintre acestea nu au fost de natură să le satisfacă doleanțele.

Printre cauzele menționate de către acestea enumerăm: nerespectarea principiilor modularizării, descompunerii funcționale și folosirii structurilor standard de control; neluarea în calcul a unor elemente specifice domeniului de aplicație; efectuarea modificărilor cu mari consumuri de resurse, lipsa unei documentații clare, coerente, complete, sistematizate și la zi etc.

Numeroase din produsele program existente în Biblioteca Națională de Programe au dovedit gradul ridicat de generalitate și adaptabilitate, ele fiind deja implementate cu sau fără modificări într-un important număr de unități economice. În măsură egală sînt solicitate produse program pe domenii de aplicație (PERSORET, GESTOC, MIFIX, CONTAB, APRODES ș.a.) sau pe tipuri de prelucrări (interogări fișiere, editări fișiere, editări rapoarte, validări date și conversie de suport ș.a.).

Pe linia preocupărilor de eficientizare a sistemelor informatice se înscriu și materialele metodologice și normative elaborate de Institutul Central pentru Conducere și Informatică și care au ca obiectiv crearea cadrului organizatoric și metodologic unitar de proiectare, realizare, organizare în biblioteci și implementare a produselor-program de uz general și totodată creșterea productivității muncii specialiștilor care activează în unitățile de informatică.

În condițiile acestui nou cadru organizatoric și metodologic crește considerabil ponderea produselor-program de uz general (tipizate) și în consecință devine tot mai necesară organizarea după criterii riguros stabilite și desfășurarea conform unei metodologii unitare a activității de implementare a produselor-program.

La stabilirea cadrului metodologic și organizatoric de desfășurare a activității de implementare a produselor-program trebuie să se aibă în vedere multitudinea de variante de implementare și anume [64]: beneficiarul precizează un anumit produs-program și cere implementarea lui ca atare; beneficiarul precizează un anumit produs-program și cere implementarea lui cu modificări; beneficiarul precizează un anumit produs-program și cere implementarea lui cu modificări care impun reproiectare; beneficiarul cere implementarea unui produs-program fără să se efectueze vreo modificare; beneficiarul cere implementarea unui produs-program cu efectuarea de modificări; beneficiarul cere implementarea unui produs-program cu modificări care impune reproiectare.

Rezultă că nu întotdeauna se poate preciza de la început varianta de implementare a unui produs-program.

De cele mai multe ori numai după parcurgerea anumitor etape ale activității de implementare se poate preciza varianta acceptată de beneficiar.

Cadrul general de desfășurare a activității de implementare prevede ca etape posibile:

a) *analiză preliminară:* se culeg informații despre specificul activității unității economice care solicită implementarea și preluarea produsului-program specificat de către aceasta;

b) *analiza de fundamentare a implementării:* se culeg informații despre specificul activității unității economice care solicită implementarea unui produs-program și se selectează produsul-program care corespunde cel mai bine exigențelor formulate de beneficiar;

c) *adaptarea produsului program:* se fac modificări pentru a se obține o nouă versiune a produsului-program corespunzător exigențelor formulate de beneficiar;

d) *implementarea propriu-zisă:* se constituie bibliotecile, fișierele, baza de date, se testează în condiții reale produsul-program, se iau măsurile organizatorice și metodologice prealabile lansării în execuție, se definitivează documentația de exploatare și se lansează în exploatare produsul-program.

În funcție de obiectivele urmărite și relațiile care se pot stabili între datele de intrare și cele de ieșire se delimitează activități premergătoare implementării și activității de implementare propriu-zisă. Dacă activitățile premergătoare implementării se desfășoară numai pentru produse-program care se preiau dintr-o bibliotecă de programe, cele de implementare propriu-zisă se desfășoară pentru orice produs-program ce urmează a fi implementat inclusiv pentru cele proiectate și realizate pe măsură (proiectate la cererea beneficiarului).

10.2. Activități premergătoare implementării

Organizarea și desfășurarea activităților premergătoare implementării unui produs-program se realizează în mai multe variante în funcție de exigențele formulate de beneficiar și măsura în care produsele-program existente în biblioteci satisfac aceste exigențe [64].

În figura 10.1 precizăm schema generală de desfășurare a activităților premergătoare implementării produselor-program.

După cum reiese din figura 10.1, *analiza preliminară* se desfășoară numai în varianta în care beneficiarul precizează produsul-program pentru care dorește implementarea și are ca obiectiv preluarea produsului-program și analiza condițiilor de implementare în vederea luării deciziei cu privire la continuarea implementării. La realizarea analizei preliminare trebuie să participe deopotrivă beneficiarul și unitatea informatică în sarcina căreia cade implementarea.

Cu această ocazie, produsul-program este identificat și executat în scop demonstrativ folosind datele de test furnizate de elaborator (odată cu depunerea lui într-o bibliotecă de programe). De asemenea, sînt analizate condițiile din unitatea economică beneficiară care vizează deopotrivă evaluarea condițiilor și implicațiilor organizatorice, analiza fluxului informațional subsistemului / aplicației pentru care se face apel la un produs-program și evaluarea resurselor materiale și umane necesare.

Rezultatele analizei preliminare sînt consemnate într-un document intitulat „NOTĂ PRELIMINARĂ IMPLEMENTĂRII", dintre care reținem îndeosebi pe cele referitoare la resursele materiale, umane și financiare necesare implementării, intrările și ieșirile, cerințe organizatorice și metodologice privind implementarea produsului program.

În varianta în care beneficiarul în comanda sa precizează doar domeniul de aplicație pentru care dorește implementarea unui produs-program este necesară desfășurarea activității de analiză în vederea fundamentării implementării..

Dintre activitățile și lucrările specifice analizei de fundamentare a implementării menționăm: analiza comenzii beneficiarului pentru a delimita cu precizie domeniul de aplicație, analiza condițiilor din unitatea economică beneficiară, analiza cataloagelor de produse-program existente în bibliotecile de programe (și dacă este cazul, selectiv sau în totalitate, a documentației de prezentare a acestora), prezentarea la beneficiar a principalelor produse-program acceptate în vederea luării deciziei în vederea implementării, pre-

luarea produsului-program (rulare demonstrativă, preluarea bibliotecilor de programe și a documentației aferente, verificarea documentației) și întocmirea documentului intitulat „NOTA DE FUNDAMENTARE A IMPLEMENTĂRII".

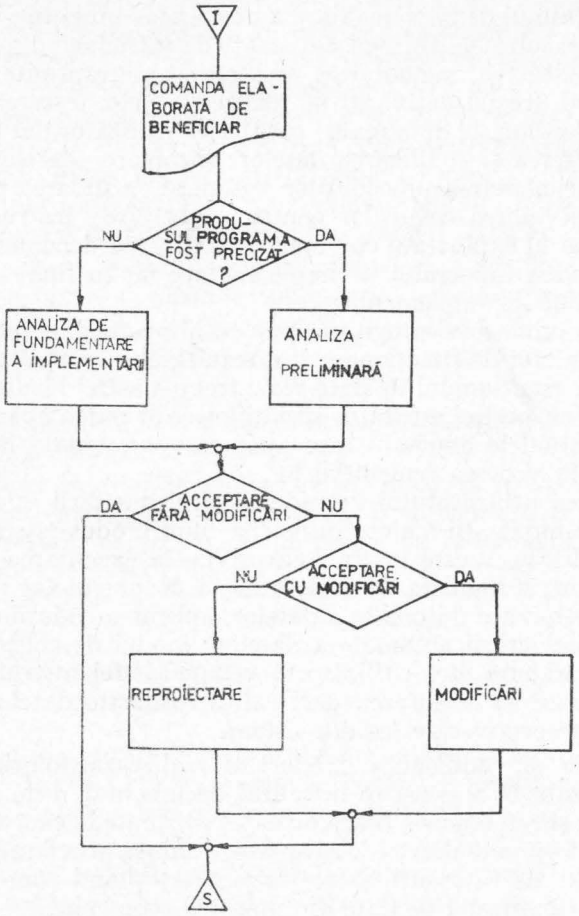


Fig. 10.1. — Schema generală a activităților premergătoare implementării produselor-program.

În cadrul documentației întocmite la finele analizei de fundamentare a implementării se face o paralelă între aria de cuprindere și organizarea generală a subsistemului / aplicației (problemei precizate de beneficiar și caracteristicile principalelor produse-program existente pe acest domeniu de aplicație).

Pe baza acestor informații beneficiarul poate opta pentru implementarea unui anumit produs-program cu sau fără modificări sau pentru reproiectarea unui produs-program întrucît cele existente nu acoperă decît parțial domeniul supus studiului.

Pentru produsele-program care nu pot fi implementate ca atare este necesară desfășurarea unei activități de modificare care are ca obiectiv adoptarea produsului-program la cerințele beneficiarului.

Prin urmare modificarea unui produs-program este similară întreținerii care a fost prezentată pe larg în capitolul 9.

10.3. Activităţi de implementare

Implementarea propriu-zisă a unui produs-program constituie o etapă de bază a procesului de implementare a produselor-program. De reuşita acestei etape depind rezultatele întregii activităţi de introducere a tehnicii moderne de calcul în culegerea, memorarea, prelucrarea şi transmiterea informaţiilor.

În cadrul acestei activităţi se realizează [64]: testarea produsului-program cu un eşantion de date reale; pregătirea utilizatorului în vederea implementării, culegerea şi codificarea datelor; încărcarea datelor în fişiere / baza de date; experimentarea procedurilor automate; validarea rezultatelor experimentării şi stabilirea costurilor pentru exploatare; instruirea personalului pentru trecerea în exploatare curentă; definitivarea documentaţiei de exploatare şi întocmirea raportului de implementare iar în final întocmirea şi avizarea raportului de implementare.

Testarea produsului-program cu un eşantion de date reale are ca obiectiv verificarea modului de funcţionare şi a rezultatelor furnizate în urma exploatării. Alegerea eşantionului de date reale trebuie astfel făcută încît să permită verificarea pe cît posibil a tuturor situaţiilor ce ar putea apărea în exploatarea curentă. Eventualele anomalii care apar se vor semnala într-o *LISTA DE ANOMALII* în vederea remedierii lor.

Pregătirea utilizatorului în vederea implementării este deosebit de importantă şi cu implicaţii majore în reuşita unui produs-program. Prin această activitate, utilizatorul este instruit cu privire la executarea procedurilor manuale; structura şi maniera de completare a documentelor primare, adaptate la cerinţele prelucrării automate a datelor, aplicarea codurilor în conformitate cu cerinţele prelucrării automate a datelor; modul de culegere a datelor din documentele primare etc. Utilizatorul trebuie astfel instruit încît să devină principalul factor în asigurarea unei calităţi ridicate datelor şi informaţiilor care intră şi respectiv care ies din sistem.

Culegerea şi codificarea datelor se realizează folosind documentaţia tipizată precizată în acest scop în cadrul documentaţiei de utilizare a produsului-program şi/sau fişierele (baza de date) existente în cadrul altor subsisteme (aplicaţii create şi actualizate de către alte produse-program). Ea se desfăşoară în exclusivitate de către utilizator (beneficiar) ţinînd cont de dispunerea în spaţiu şi timp a surselor de date din unitatea economică.

Încărcarea fişierelor (bazei de date) presupune participarea deopotrivă a utilizatorului şi a unităţii de informatică. În cadrul acestei activităţi folosind proceduri automate şi manuale se realizează validarea datelor şi încărcarea celor corecte în fişierele (baza de date) a sistemului. În practică procedurile de validare date şi conversie de suport editează *liste de control* care conţin informaţii cu privire la erorile detectate, articolele şi cîmpurile în care acestea apar şi datele care au fost luate în calcul. În baza listei de control sînt detectate erorile şi semnalate, prin verificare cu datele culese, datele neluate în calcul. Sistemele cu lucru în regim interactiv sînt astfel construite încît permit utilizatorului să efectueze direct corecturi, lista de control fiind doar document justificativ pentru a proba că datele au fost înregistrate.

Experimentarea procedurilor automate are ca obiectiv verificarea programelor în condiţii de exploatare cu date reale. În condiţii de experimentare se realizează în paralel prelucrarea manuală a datelor şi prelucrarea în condiţiile folosirii produsului-program. Prin compararea rezultatelor obţinute prin cele două variante de prelucrare se pot detecta eventualele erori, stabili

cauzele și lua măsurile de eliminare. O importanță deosebită se acordă realizării actualizării fișierelor (bazei de date). Cînd produsul-program este de complexitate mare se indică experimentarea treptată ținînd cont de structura generală a produsului-program și interdependența între principalele proceduri / module.

Erorile și anomaliile detectate în faza de experimentare sînt semnalate în FISA DE ANOMALII și analizate (de către echipa de implementare sau de către echipa de întreținere) în vederea eliminării lor.

Validarea rezultatelor experimentării și stabilirea costurilor pentru exploatare se realizează cu participarea directă atît a beneficiarului cît și a unității de informatică ce are în sarcină implementarea produsului-program.

Rezultatele obținute în faza de experimentare sînt comparate cu cele obținute prin prelucrare manuală pentru a fundamenta avizul de implementare.

Cu această ocazie se stabilesc costurile necesare exploatării curente a produsului-program și se determină indicatorii de eficiență corespunzători.

Instruirea personalului pentru trecerea în exploatare curentă vizează deopotrivă utilizatorul și personalul de exploatare din unitatea de informatică.

Tot în cadrul acestei activități se întocmește graficul pentru exploatarea curentă a produsului-program.

Definirea documentației de exploatare se realizează în baza informațiilor culese în fazele anterioare. În această etapă se întocmește și RAPORTUL DE IMPLEMENTARE, care conține informații de fundamentare a avizării implementării produsului-program.

10.4. Aspecte organizatorice

Faptul că tot mai multe unități economice apelează la produse-program de uz general a dus la creșterea volumului activității de implementare în cadrul activităților desfășurate în unitățile de informatică.

În consecință, s-a impus delimitarea ca activitate distinctă „implementarea produselor-program existente în biblioteci de programe" și desfășurarea ei în cadrul unui organism funcțional corespunzător care în practică poate fi un atelier structurat pe funcții specializate în implementarea unor anumite categorii de produse-program.

În figura 10.2 se prezintă organizarea de principiu a atelierului de implementare a produselor-program, cu gruparea formațiilor de lucru pe cele două tipuri de produse-program și anume: produse-program pe tipuri de operații de prelucrare; produse-program pe domenii de activitate și activitățile din cadrul acestora.

Se recomandă ca formațiile de implementare a produselor-program să fie formate din cei mai experimentați analiști și programatori care au participat la proiectarea și realizarea de produse-program din aceeași categorie sau din categorii apropiate. Totodată un rol activ revine utilizatorului care are sarcini pe linia completării documentelor primare corespunzătoare cerințelor impuse de folosire a unui anumit produs-program, verificării și transmiterii acestora, stabilirii procedurilor de control în vederea validării datelor la introducerea în sistem și pe parcursul prelucrării, utilizării corecte a costurilor și descrierii structurii și a conținutului informațional al situațiilor de ieșire.

Prin urmare, aspectele organizatorice privind implementarea produselor-program privesc deopotrivă utilizatorul și informaticianul; ele referențiază structurile funcționale corespunzătoare și cadrul metodologic de desfășurare și formare a cadrelor.

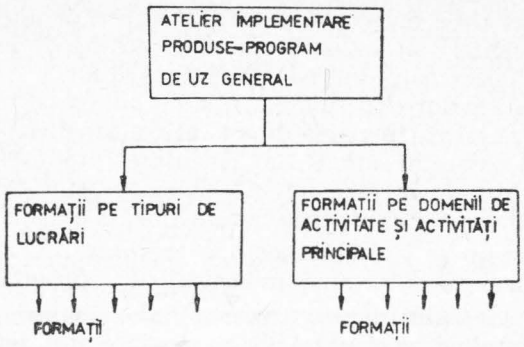


Fig. 10.2. — Organizarea atelierului de implementarea produselor-program.

Odată cu derularea activității de implementare a produsului-program se definitivează și documentația de utilizare, de exploatare și de realizare (în cazul cînd acestea au suferit modificări față de versiunea preluată dintr-o bibliotecă de programe). Dacă asupra produsului-program s-au efectuat modificări, rezultatele obținute pot fi apreciate ca o nouă versiune a produsului-program și inclus ca atare într-o bibliotecă.

11. Întreţinerea programelor

11.1. Scopul întreţinerii programelor

Din cele prezentate în capitolele precedente a rezultat că elaborarea unui program este făcută pe baza specificaţiilor de definire funcţională întocmite de utilizator prin care sînt definite intrările, ieşirile şi algoritmii de obţinere a ieşirilor pornind de la intrări.

Între momentul definirii problemelor în vederea prelucrării automate a datelor şi cel al utilizării curente şi chiar pe parcursul acesteia din urmă apar, de cele mai multe ori, modificări care trebuie la rîndul lor operate în program.

Modificările vizează în primul rînd logica şi datele problemei. Ele mai pot fi cerute însă de modificări ce apar în plan informatic al tehnicilor şi instrumentelor de rezolvare a problemei, al configuraţiei de calcul, al sistemului de operare.

Iată de ce programele fac obiectul unei activităţi de întreţinere care are ca prim obiectiv efectuarea de modificări în cadrul programelor pentru a răspunde cerinţelor la zi ale beneficiarului şi condiţiilor concrete de operare.

Aşa cum se menţionează în literatura de specialitate costul întreţinerii programelor devine critic. O mare parte din efort este dedicat întreţinerii programelor în opoziţie cu dezvoltarea de noi aplicaţii.

Întreţinerea programului înseamnă atît dezvoltare şi documentare cît şi depanare. După caz ea poate include şi serviciile de instruire cu privire la utilizarea în practică a unui produs-program.

Iată pentru ce considerăm tot ca obiective ale activităţii de întreţinere a programelor depanarea şi documentarea acestora.

Mult timp s-a crezut că un program este o „cutie neagră" ce permite funcţionarea nelimitată în timp, fără modificări. În realitate însă nu există program fără erori.

Pe de altă parte orice program este modificat continuu în decursul perioadei de utilizare: se adaugă noi funcţii, se extinde capacitatea de lucru etc.

Se estimează că 60 ÷ 70% din cheltuielile totale pentru un produs-program revine întreţinerii produsului după livrare.

Aceste cheltuieli se pot reduce substanţial dacă se face un efort suplimentar de elaborare în fazele de dezvoltare ale produsului.

După părerea noastră termenul „întreţinerea" programelor a fost în mod greşit asimilat întreţinerii echipamentelor. Se ştie că întreţinerea echipamentelor (hardware-ul) înseamnă reparare, adică readucerea unor piese la starea anterioară căderii. În domeniul programelor, însă, căderea (failure) — aşa cum se arată în [65] — este determinată de apariţia unei stări incorecte existente, ceea ce înseamnă că o reparare schimbă starea anterioară.

Programele, nefiind ceva fizic, nu se degradează prin utilizare şi nici nu se deteriorează în timp.

Dacă un program este corect, el va rămîne corect — dacă nu se fac modificări — cît timp va fi utilizat.

Pe timpul utilizării programelor sînt descoperite erori și eliminate, ceea ce duce implicit la creșterea fiabilității.

În consecință, spre deosebire de echipament starea programului se îmbunătățește în timp.

Repararea erorilor de codificare deține o pondere mică în cadrul activității de întreținere post-lucrare.

În [66] se estimează că 30% din discrepanțe în timp sînt datorate erorilor de codificare iar 70% din greșelile de proiectare sau specificații de definire.

Iată și motivul pentru care vom considera că un sistem este flexibil dacă este astfel elaborat încît să permită preluarea schimbărilor fără a construi sau înlocui întregul sistem. Pe cuprinsul acestei lucrări, prin întreținerea programelor vom înțelege deci modificarea stării anterioare fie pentru punerea lui în concordanță cu specificațiile de definire a problemei la zi, fie pentru eliminarea unor erori depistate în exploatare, fie pentru ambele cazuri.

Prin urmare, modificările ulterioare asupra unui program sînt realizate din următoarele considerente [67]: programul conține erori care trebuie eliminate, caz în care întreținerea înseamnă *corecții* asupra programului; programul nu mai este în concordanță cu mediul în care se execută, caz în care întreținerea înseamnă *adaptare* la noile condiții de execuție; se cere îmbunătățirea parametrilor calitativi ai programului, caz în care întreținerea înseamnă *perfecționare*.

Experiența practică demonstrează că modificarea programelor este un proces complex în care se pot strecura erori cauzate îndeosebi de lipsa unei documentații de întreținere sau existența ei într-o formă neadecvată. De asemenea, persoanele care realizează întreținerea, de cele mai multe ori, nu sînt și cele care au elaborat programul.

11.2. Cerințe în întreținerea programelor

Pentru desfășurarea în condiții de eficiență a întreținerii programelor trebuie ca încă din faza de concepere a acestora să se prevadă posibilitățile de modificare. De asemenea, trebuie prevăzute și incluse în program posibilitățile de extindere cu noi funcții.

În acest scop trebuie stabilit [68]: ce limite și capacități se vor putea schimba; care este echipamentul sau resursele ce se vor modifica; ce parametri sînt ficși și care sînt modificabili; ce capacitate de rezervă trebuie asigurată pentru fiecare resursă etc.

O problemă importantă ce trebuie avută, de asemenea, în vedere în faza de concepere a programului este interfața cu utilizatorul. Practica arată că interfața cu utilizatorul suferă întotdeauna schimbări. Programul trebuie astfel conceput încît să poată fi adăugate noi comenzi utilizator sau modificate formatele celor existente fără a influența restul comenzilor (modulelor).

Cheia modificării ușoare a programelor este *claritatea*, *simplitatea* și *documentarea* în elaborarea acestora.

Pentru aceasta este necesar să se respecte principiile ingineriei programării și să se aplice tehnicile evoluate în programare și anume: modularizare; descompunerea funcțională; utilizarea structurilor standard de control; co-

dificarea structurată; alocarea cerințelor funcționale elementelor de cod; documentarea programelor ș.a.

Pentru realizarea în condiții de eficiență a activității de întreținere a programelor se fac următoarele recomandări:

— modulele să fie structurate pe nivele ierarhice;

— fiecare modul să realizeze o funcție din cadrul problemei supusă studiului;

— un modul să conțină un număr de instrucțiuni limitat dacă este posibil la o pagină de listing;

— un modul să aibă o singură intrare și o singură ieșire;

— interfețele cu utilizatorul să fie localizate în cîteva rutine;

— să se reducă pe cît posibil interfața dintre module;

— organizarea internă a modulului să se facă în conformitate cu principiile ingineriei programării și utilizării structurilor standard de control;

— fiecare program trebuie să conțină suficiente informații pentru a ușura înțelegerea și adaptarea la noi condiții (se apreciază că autodocumentarea programelor trebuie să devină o tehnică obligatorie în elaborarea produselor-program);

— să se folosească metode și instrumente corespunzătoare de localizare automată a erorilor și de realizare tot automată a modificărilor cu toate influențele pe care le au acestea;

— să se folosească instrumente sub forma produselor-program care să permită inspectarea programelor și/sau înlocuirea unui identificator cu un alt identificator ori unde acesta apare în program;

— să se folosească tehnici mai eficiente în programare conform cărora anumite descrieri de date sau de algoritmi de prelucrare să poată fi memorate în biblioteci de programe și apelate ori de cîte ori este nevoie.

În sistemele mari un rol deosebit îl are controlul și structurarea datelor locale și globale. Metoda cea mai bună de a controla și manipula datele pe parcursul conceperii și codificării produselor-program este gruparea lor într-un modul și definirea simbolică pentru referirea lor. De asemenea, este necesar ca structurile de date să fie documentate și eventual să existe un dicționar cu identificatorii datelor folosite într-un produs-program.

O altă cerință majoră în întreținerea programelor o constituie existența unei documentații de calitate. Ea presupune atît manuale de realizare (organigrame, descrieri de programe, listinguri etc.) cît și de utilizare destinată beneficiarului. Documentația trebuie la rîndul ei întreținută adică ținută la zi, prin urmare actualizată odată cu modificările din program. În caz contrar, oricît de bine documentat ar fi un produs-program, toată documentația lui își pierde valabilitatea dacă nu reflectă starea actuală a programului.

11.3. Modificarea programelor

Din cele prezentate în paragrafele precedente a rezultat că programele se corectează, adaptează și/sau perfecționează pe durata ciclului lor de viață iar pentru aceasta se impune ca ele să fie construite astfel încît să faciliteze aceste activități, altfel spus să fie ușor modificabile.

Literatura de specialitate [65] menționează ca obligatorie cuprinderea în tema de elaborare a unui produs-program a modificabilității ca un scop al proiectării.

Beneficiarul poate cere ca furnizorul să se supună unui plan de dezvoltare care să cuprindă tehnicile folosite pentru a construi un produs-program modificabil.

În [65, 66, 69] se arată că criteriile de alegere pentru produse-program, modificabile, trebuie să aibă la bază următoarele idei:

a) Utilizarea unui produs-program care nu a fost livrat trebuie evitată, iar utilizarea unui produs-program care nu a fost construit sau documentat pentru a fi întreţinut trebuie examinată cu prudenţă.

b) Utilizarea limbajelor de asamblare în locul unui limbaj de nivel înalt trebuie studiată cu atenţie şi în cele mai multe cazuri nerecomandată.

c) Însoţirea planului de dezvoltare a produsului-program de un plan suport de întreţinere ulterioară în ciclul de dezvoltare, în care să figureze cerinţele post-livrare pentru modificarea programelor.

d) Furnizarea unei documentaţii adecvate pentru utilizarea în activitatea de întreţinere post-livrare.

e) Furnizorul trebuie să fie obligat prin contract să livreze toate procedurile şi materialele de test de acceptanţă, inclusiv rezultatele testelor care pot fi păstrate pentru a fi efectiv folosite în timpul întreţinerii post-livrare pentru a asigura ca modificările să nu fi degradat funcţiile de bază ale produsului-program.

Este cert că a fost depăşită perioada cînd important era doar ca un produs-program să funcţioneze. Acum se cere pe lîngă funcţionare în condiţii de eficienţă şi să fie uşor modificabil.

Se impune deci ca orice produs-program să fie astfel elaborat încît să fie uşor de întreţinut adică modificabil.

Lista cu formatul sursă al produsului-program trebuie să constituie suportul principal pentru activitatea de întreţinere. Pentru aceasta trebuie introdusă o disciplină în dezvoltarea de produse-program pentru a asigura o comentare adecvată a listei sursă.

În [65] sînt evidenţiate următoarele criterii pentru redactarea listei cu codul sursă pentru ca documentaţia să satisfacă cerinţele întreţinerii:

a) Fiecare modul de program să conţină un bloc antet cu comentarii care descrie: numele programului; data ultimei revizii; cerinţe cu privire la precizie, scop, limitări şi restricţii; istoria modificării (o listă a schimbărilor şi data realizării lor); intrările şi ieşirile; metode folosite sau definiţii de algoritmi; tipurile de erori sau proceduri de terminare în caz de eroare.

b) Să fie specificate prin comentarii comunicările între module.

c) Să fie comentate punctele de decizie şi alternativele de salt şi utilizată alinierea pentru a arăta structura pe blocuri a logicii codului.

d) Să fie explicată prin comentarii logica implementării pentru fiecare algoritm.

e) Să fie definită prin comentarii logica implementării fiecărui algoritm.

f) Să fie comentate domeniile parametrilor şi condiţiile implicite pentru aceştia.

g) Comentariile să permită selectarea unor valori specifice pentru intrări pentru a face posibilă trecerea testelor specializate.

i) Să fie descrise suficient de detaliat funcţia fiecărui modul şi intrările/ieşirile pentru a facilita testarea.

j) Toate instrucţiunile-program care au suferit modificări trebuie să aibă un număr de identificare inclus care le asociază cu modificarea.

h) Introducerea informaţiilor cu privire la impactul unei schimbări în alte porţiuni ale programului.

Aceste criterii vizează ca obiectiv scrierea codului astfel încît să conțină suficiente informații pentru a determina sau verifica obiectivele codului, restricțiile, intrările, ieșirile, componentele sau starea reviziei (versiunii).

Cu toate acestea trebuie să facem precizarea că listingul nu este suficient pentru a asigura modificabilitatea produselor-program. Sînt necesare și descrieri de ansamblu ale produsului-program, organizate după principii operaționale, cu specificarea fluxurilor pentru fiecare task operațional.

Ori de cîte ori o problemă este notată, în timpul întreținerii se cere verificat listingul programului care referă rutinele și datele implicate. Numai după aceea se trece la analiza listingurilor programelor implicate pentru informații de detaliu.

Este foarte important ca activitatea de modificare a unui produs-program post-livrare să se desfășoare în conformitate cu principiile care au stat la baza proiectării și implementării utilizate de cel ce a elaborat produsul-program și să respecte întrutotul etapele elaborării produselor-program.

11.4. Instrumente de ajutor în organizarea și întreținerea programelor

Dificultățile care apar în practica întreținerii produselor-program cu grad ridicat de complexitate au determinat preocupări pe linia elaborării unor produse-program cu rol de gestionare a bazei de programe în vederea unei întrețineri mai ușoare.

Dintre acestea evidențiem îndeosebi procedurile de manipulare a bazei de programe AMSBAS dezvoltate de A. Becea [70] în cadrul Institutului pentru Tehnică de Calcul pentru gestionarea și întreținerea bazei de date AMS *) pentru a asigura:

— păstrarea ordonată a programelor sistemului de operare AMS;

— evidența strictă a corecțiilor, a versiunilor acestor programe, bazată pe principiul autoidentificării programelor și a modulelor lor;

— creșterea eficienței operațiilor de dezvoltare și de întreținere a programelor AMS;

— ordonarea activităților de elaborare și de întreținere de programe pentru un sistem de programe cu componente generate diferit în funcție de o serie de opțiuni standard;

— asigurarea unei alimentări eficiente a discurilor de generare AMS cu ultimele versiuni ale componentelor solicitate în funcție de anumite opțiuni standard.

Procedurile AMSBAS reprezintă un ansamblu de fișiere de comenzi care permit descrieri și tratări adecvate de entități (module, programe etc.) memorate în baza de programe, asigurînd rapidizarea, simplificarea și standardizarea diverselor operații de prelucrare executate de utilizator.

Ideea de bază este repertorizarea entităților în cataloage, în care se memorează atît entitatea furnizată de utilizator cît și caracteristicile acesteia. În plus AMSBAS asigură proceduri standard de prelucrare a entității utilizator (asamblare, catalogare, editare de legături, corectare etc.) prin comenzi ce se reduc, de cele mai multe ori, la răspunsuri de tip DA sau NU.

---

*) AMS, sistem de operare elaborat în cadrul ITC pentru minicalculatorul Independent.

Procedurile AMSBAS se împart în mai multe categorii:

a) proceduri de inventariere: OPECAT — creare/manipulare cataloage, OPEPROC — repertorizare programe, OPEMOD — repertorizare module din program;

b) proceduri de descriere entități: PROG — definire programe, MOD— definire module din programe, VERS — definire versiuni de programe;

c) proceduri pentru descrierea modului de lucru: OPT — definire opțiuni, GRUP — definire grupe de nume, IMP — definire valori implicite;

d) proceduri de prelucrare: EDI — editare module, COR și ACT — editare și asamblare module, ASM — asamblare module, LNK — editare de legături, GEN — transfer entitate pe discul de generare, MAC — prelucrare macroinstrucțiuni.

Toate aceste proceduri se află într-un catalog supervizor unic și sînt utilizate simultan de mai mulți utilizatori. Acest catalog este identificat în procedurile AMSBAS prin PD: [2, 2] unde PD este un periferic logic, ce trebuie asignat discului fizic pe care se află catalogul supervizor.

Caracteristicile ce se păstrează pentru fiecare program în AMSBAS sînt următoarele:

— Nume program: identifică programul la nivel de catalog.

— Varianta de program: permite diferențierea între diferite variante ale aceluiași program.

— Indicatorul de program privilegiat.

— Numele modulelor sursă.

Un program poate avea mai multe categorii de module sursă:

- module proprii: descrise în dicționarul de module asociat programului și compilate pe contul acestuia;
- module multi-program: descrise în dicționarul de module asociate programului dar și în dicționare de module ale altor programe (se compilează diferit pentru fiecare program);
- module împrumutate: module proprii sau multiprogram aparținînd altor programe din același catalog, ce sînt referite la editarea de legături;
- module de biblioteci comune utilizator;
- module din biblioteca sistem (SYSLIB).

— Fișierul prefix de compilare: fișier de tip MACRO folosit la compilarea modulelor proprii și a modulelor multiprogram.

— Numele general al bibliotecii obiect. Programul are o colecție de biblioteci obiect, corespunzătoare combinațiilor de opțiuni standard: limbă (română, engleză), unitate de translatare a adreselor (MMU) și inserare sau nu de puncte de depanare. Manipularea bibliotecilor se face automat de către AMSBAS.

— Generatorul comenzilor de editare de legături: fișier de comenzi definit de utilizator pentru fiecare program executabil care generează fișierul de comenzi folosit de editorul de legături pentru a crea imaginea executabilă a programului.

Un modul al unui program are următoarele caracteristici:

— Numele modulului.

— Numele programului căruia îi aparține. Modulele proprii sînt asociate unui singur program și descrise numai în dicționarul de module al acestui program. Modulele multiprogram sînt asociate cu mai multe programe trebuind să fie descrise în dicționarele de module ale fiecăruia dintre programe.

— Indicatorul de modul condiționat de limbă: arată dacă modulul conține mesaje ce trebuie compilate diferit în funcție de limba în care se dorește programul (engleză sau română).

— Indicatorul de modul condiționat de program: arată dacă modulul este de tip multiprogram, fiind repartizat în dicționarele de module ale mai multor programe și compilat cu prefixe de compilare distincte pentru fiecare din acestea.

— Numele de grup multiprogram (numai pentru modulele multiprogram).

— Indicatorul de modul condiționat de MMU (numai pentru modulele din programele privilegiate): arată dacă modulul conține secvențe de instrucțiuni compilate diferit după cum modulele obiect sînt destinate de a fi executate pe sisteme de calcul cu sau fără dispozitivul de translatare a adreselor.

În fiecare catalog se păstrează un dicționar de versiune unic, care descrie istoria generală a tuturor corecțiilor efectuate asupra modulelor din catalog. Versiunile prin care trece fiecare modul sursă sînt memorate și în istoria de corecții ce se introduce automat la începutul fiecărui modul. Procedurile AMSBAS permit ca istoria generală de corecții din dicționarul de versiuni ca și istoriile de corecții de la începutul modulelor să fie păstrate numai în interiorul unei serii de corecții, fiind reinițializate la trecerea la o nouă serie de corecții.

Pentru fiecare versiune se memorează următoarele informații: identificatorul de versiune (o literă urmată de 3 cifre zecimale). (Versiunile succesive sînt numite prin identificatori în ordine crescătoare); autorul corecției; data corecției; descriptorul corecției (una sau mai multe linii în care se prezintă în mod sumar corecția efectuată); lista de module corectate.

Descriem mai jos un mod general de lucru cu procedurile AMSBAS;

Un utilizator dorește să introducă un nou produs-program în baza de programe, își creează un catalog propriu (dacă este cazul), după care, cu:

— Procedura OPECAT: inițializează la nivel de catalog toate dicționarele de programe, versiuni, opțiuni etc. De asemenea, utilizatorul furnizează unele informații referitoare la versiunea programului: identificatorul de versiune, autor, descriere versiune etc.

— Procedura PROG: utilizatorul își definește programul în dicționarul de programe. Pentru aceasta furnizează informația pentru descrierea programului: numele programului, varianta, dacă este privilegiat sau nu, numele fișierului prefix de compilare, numele bibliotecii obiect etc.

— Procedura OPEPROG: inițializează dicționarul de module, biblioteci obiect și definește fișierul prefix de compilare și generatorul fișierului de comenzi pentru editarea de legături.

— Procedura MOD: definește modulele ce alcătuiesc programul. Se specifică numele modulelor și dependența de limbă și de unitatea de translatare a adreselor MMU.

— Procedura OPEMOD: copiază modulele sursă (aflate eventual pe alt dispozitiv) în catalogul curent al utilizatorului. După copiere utilizatorul inițializează istoriile de corecții ale tuturor modulelor.

— Procedura ASM: asamblează modulele sursă ale programului (toate sau selectiv) și cataloghează automat modulele obiect în bibliotecile corespunzătoare opțiunilor specificate (limba română/engleză, cu/fără MMU, cu/fără puncte de depanare).

— Procedura LNK: face editarea de legături a programului creind imaginea executabilă.

— Procedura GEN: transferă bibliotecile obiect cu fișierele de editare de legături pe discul de generare.

Procedurile de prelucrare (EDI, COR, ASM, LNK) pot fi apelate în mod repetitiv și independent pînă la obținerea programului corect.

Trebuie spus că, fiecare din procedurile enumerate mai sus asigură și alte facilități utilizatorului. De exemplu, procedura OPEPROG asigură și listarea dicționarului de module, a fișierului prefix de compilare sau a generatorului de fișiere de comenzi pentru editarea de legături, compactarea bibliotecilor obiect ale programelor, înlănțuire dicționar de module pentru a face accesul mai rapid etc. Procedura MOD asigură și ștergerea definirii unui modul, modificarea definirii unui modul, listarea definirilor de module etc.

În condițiile în care aplicațiile utilizatorului devin tot mai complexe, la care lucrează un grup tot mai mare de programatori, procedurile AMSBAS se înscriu pe linia ușurării prelucrării acestor programe prin standardizarea și automatizarea manipulării lor. Problematica întreținerii programelor găsește rezolvare eficientă în cadrul de cercetare-proiectare de genul atelierelor de programare.

12. Calitatea produselor-program

12.1. Orientări și realizări în măsurarea calității produselor-program

Eforturile actuale îndreptate spre crearea și consolidarea unei industrii de programe impun totodată intensificarea preocupărilor care au ca obiectiv creșterea calității produselor-program.

Similar calității produselor din domeniile producției materiale, calitatea produselor-program exprimă gradul în care ele satisfac o nevoie socială în funcție de parametrii tehnico-economici, gradul de utilitate și eficiență economică în exploatare [83]. Pentru a fi de calitate un produs-program trebuie să aibă anumite caracteristici.

Deși preocuparea pe linia obținerii unor produse-program de calitate a existat de la începuturile programării stabilirea unui sistem de caracteristici și criterii care permit exprimarea acesteia datează de la începuturile anilor 1970. În anul 1972 s-a întrunit la Centrul de Programare al firmei I.B.M. de la San Jose un grup de studiu pentru cercetarea noțiunii de calitate a programelor și stabilirea unor criterii sau unități de măsură a ei.

La început calitatea a vizat îndeosebi absența erorilor ce ar provoca oprirea execuției programului sau comportarea lui imprevizibilă. Dar măsurarea calității unui program numai prin numărul de erori pe care le conține cere utilizarea programului o perioadă îndelungată de timp, perioadă în care sînt înregistrate toate erorile apărute în cursul exploatării.

Important este însă de a măsura calitatea programului înainte de livrarea lui la utilizator, și mai mult, chiar pe etape ale procesului de elaborare.

Limitînd însă calitatea la numărul de erori care apar în procesul execuției estimarea calității pe parcursul și îndeosebi în momentul obținerii produsului-program pare imposibilă.

Problema era deci de a găsi o metodă de măsurare a erorilor pe faze ale elaborării produselor-program.

Astfel se pot număra erorile descoperite în fazele de elaborare ale programului și se poate aprecia eficiența eliminării erorilor în aceste faze. De asemenea, presupunînd cunoscute erorile ce vor fi descoperite după livrarea produsului se poate defini raportul de eficiență cumulativă de eliminare a erorilor [25], și anume:

$$EC = \frac{\text{nr. erori eliminate înainte de livrare}}{\text{nr. erori totale}}.$$

Raportul EC a fost măsurat pentru mai multe produse software ale firmei I.B.M. după o perioadă îndelungată de funcționare la utilizator. Rezultatele obținute arată că pentru 1 000 de linii sursă există, în medie, 30÷35

de erori, iar raportul EC este de aproximativ 90÷95%. Astfel pentru un program de 15 000 de linii sursă ar trebui să existe cel puțin 450 de erori, și considerînd o eficiență cumulativă de 95% înseamnă că după livrarea produsului mai rămîn cel puțin 23 de erori sau aproximativ 1,5 erori pe 1000 linii sursă. Pentru evaluarea calității pe baza erorilor eliminate s-au propus diferite metode dintre care menționăm metoda lui Remus și Ziller [26].

Metoda lui Remus și Ziller de evaluare a calității programelor

Metoda lui Remus și Ziller de evaluare a calității unui program se bazează pe numărul de erori eliminate în fazele elaborării programului.

Se presupune că fiecare program (sau modul) conține inițial o mulțime de erori numite erori inițiale — notate e — care sînt introduse în prima versiune de elaborare a programului. Se presupune, de asemenea, că există o secvență de etape de verificare a programului în cursul cărora se descoperă mai multe erori și se corectează. În cadrul fiecărei etape se elimină o parte din erorile inițiale, dar se introduc și alte erori din cauza unor eliminări incorecte. Fie următoarele notații: e — numărul de erori inițiale; e_i — numărul de erori ce există la trecerea la etapa i; d_i — raportul de erori detectate la etapa i; r_i — raportul de erori incorect eliminate la etapa i.

Numărul de erori rămase după etapa i este:

$$e_{i+1} = (1 - d_i)e_i + r_i d_i e_i = (1 - d_i(1 - r_i))e_i,$$

$(1 - r_i)$ poate fi văzut ca raportul de erori eliminate corect la etapa i, iar $d_i(1 - r_i)$ este fracția de erori efectiv scoase la etapa i.

Se presupune că etapele de eliminare a erorilor sînt cele ale procesului de corecție aplicat de firma IBM: (1) inspecția și revizuirea codului, efectuate de cei ce elaborează programul; (2) testarea programului, efectuată de alte persoane. (Corecția erorilor descoperite în cadrul acestor două etape se face de către elaboratorii programului); (3) livrarea la beneficiari a produsului și înregistrarea erorilor descoperite de utilizator. Aceste erori sînt corectate de un grup de serviciu special (Field Engineering Division of IBM).

Folosind noțiunile precedente avem: $f_1 = d_1 e$ este numărul de erori detectate în etapa 1; $g_1 = (1 - r_1)d_1 e$ — numărul de erori eliminate în etapa 1; $f_2 = d_2(e - f_1 + r_1 d_1 e) = d_2(1 - d_1(1 - r_1))e$ — numărul de erori detectate în etapa a 2-a; $g_2 = (1 - r_2)f_2 = (1 - r_2)d_2(1 - d_1(1 - r_1))e$ — numărul de erori eliminate în etapa a 2-a.

În figura 12.1 se reprezintă grafic numărul de erori eliminate în primele două etape. Numărul de erori rămase în produsul livrat utilizatorului este $e - g_1 - g_2 \geqslant e - f_1 - f_2$.

Problema este de a estima numărul erorilor rămase în produsul livrat utilizatorului. Dacă s-ar cunoaște numărul total de erori t ce apar în decursul utilizării produsului atunci numărul de erori rămase în produs la livrarea lui este

$$u_2 = t - f_1 - f_2,$$

unde f_1 și f_2 se scad direct pentru că erorile introduse prin corecțiile din etapele 1 și 2 sînt cuprinse în t.

Estimarea numărului total de erori de detectat în toată perioada de utilizare a programului este

$$t = e + i,$$

unde i este numărul de erori introduse la eliminarea altor erori, în decursul tuturor etapelor de corectare ale programului.

Presupunem că raportul r_i de erori introduse este constant (r). În acest caz:

$$i = re + r^2e + \dots r^ne,$$

$$t = e(1 + r + r^2 + \dots r^n) = e(1 - r^{n+1}) / (1 - r).$$

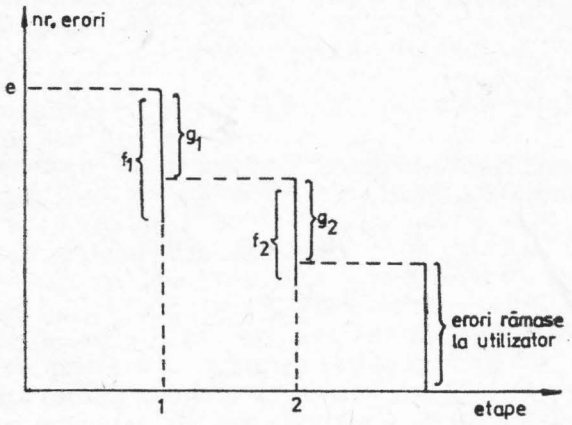


Fig. 12.1. — Grafic cu număr de erori eliminate.

La limită, cînd n tinde la infinit, avem:

$$t = e/(1 - r).$$

Presupunem că d_i este constant (d) pentru primele două etape de eliminare a erorilor. Eficiența eliminării erorilor pentru primele două etape este

$$K = (1 - r)d.$$

Acest rezultat se poate obține și direct. De exemplu, pentru faza 1:

$$K_1 = \frac{f_1}{t} = \frac{de}{e/(1 - r)} = (1 - r)d.$$

Deoarece eficiența eliminării erorilor este K atunci $1 - K$ este rata erorilor rămase nedetectate. Deci numărul erorilor rămase după etapa n este

$$u_n = t(1 - K)^n,$$

de unde

$$u_2 = t(1 - K)^2.$$

Deoarece K nu se poate măsura se introduce un nou factor ce poate fi măsurat

$$\mu = \frac{f_1}{f_2} = \frac{d_1e}{d_2(1 - d_1(1 - r))e}.$$

Presupunem $d_1 = d_2 = d$, deci

$$\mu = \frac{1}{1-(1-r)d} = \frac{1}{1-k} \quad \text{sau} \quad 1-k = \frac{1}{\mu}.$$

Aceasta înseamnă că numărul erorilor rămase după fiecare etapă de eliminare a erorilor este $1/\mu$ din numărul de erori prezente la intrarea în etapă. Creșterea lui μ implică îmbunătățirea calității programului. Aceasta se poate vedea din formula lui u_2

$$u_2 = \frac{t}{\mu^2}$$

Totuși această ecuație nu este practică pentru că nu se cunoaște t. Se poate estima t cu ajutorul lui f_1 astfel

$$t = \frac{f_1}{k} = \frac{f_1}{1-\frac{1}{\mu}} = \frac{f_1\mu}{\mu-1},$$

de unde

$$u_2 = \frac{f_1}{\mu(\mu-1)}.$$

Dacă se consideră $f_2 = \frac{f_1}{\mu}$, rezultă

$$u_2 = \frac{f_2}{\mu-1}.$$

Aceste ecuații sugerează un alt factor de măsurare a eficienței de eliminare a erorilor: eficiența de eliminare cumulativă ce reprezintă fracția de erori eliminate înainte de livrarea produsului

$$\text{EC} = \frac{t-u_2}{t} = \frac{t-\frac{t}{\mu^2}}{t} = 1-\frac{1}{\mu^2}.$$

Dacă EC este apropiat de 1 atunci puține erori vor fi eliminate după livrarea produsului. În acest caz atenția trebuie să se îndrepte spre minimizarea numărului de erori inițiale (e). Dacă presupunem că $d = 1$, atunci la fiecare etapă se descoperă toate erorile; atunci formulele devin:

$$f_1 = e,\ f_2 = re,\ k = 1-r,\ \mu = \frac{1}{1-(1-r)} = \frac{1}{r},\ u_2 = tr^2,\ \text{EC} = 1-r^2.$$

Aceasta arată că există o limită a nivelului de calitate ce ar putea fi atinsă prin detectarea tuturor erorilor din program. Formulele precedente au fost dezvoltate utilizînd numărul total de erori dintr-un program. Ele rămîn valabile pentru mărimi normalizate, ca de exemplu numărul de erori pe 1 000 linii de cod sursă (1KL). Astfel indexul de calitate QI, care este calculat pentru produsele software, se normalizează la KL în decursul unui an. Dacă

se notează cu v estimarea duratei de viață (în ani) a produsului, atunci indexul de calitate este

$$QI = \frac{u_2}{v} = \frac{f_1}{\mu v(1 - \mu)},$$

unde f_1 este măsurat în număr de erori/KL.

De exemplu, pentru a avea un index de calitate QI de 0,6 erori pe KL-an, o mare cantitate de erori trebuie descoperită înainte de livrarea produsului. Dacă se presupune o perioadă de utilizare a produsului de 4 ani, atunci numărul de erori u_2 eliminate înainte de livrarea produsului trebuie să fie mai mic de 2,4 pe KL.

Dacă se presupune că $t = 40$/KL, atunci din formula $u_2 = t/\mu^2$ se obține: $2{,}4 = 40/\mu^2$ sau $\mu \simeq 4$.

Ținînd cont că $\mu = f_1/f_2$ aceasta înseamnă că numărul de erori descoperite în faza de dezvoltare trebuie să fie de 4 ori mai mare decît cele detectate în faza de testare. Dacă QI = 0,05 atunci μ trebuie să fie 14.

Invers, dacă se numără erorile detectate în fazele de concepere și testare (f_1 și f_2) se poate aproxima μ și dacă se estimează durata de viață a produsului se poate aprecia calitatea produsului prin calcularea indexului de calitate QI. Acesta este de fapt interesul formulelor dezvoltate mai sus.

În concluzie, prin cele prezentate se arată cum, pe baza unor măsurători posibile, se poate estima numărul de erori ce vor fi descoperite în perioada de utilizare a programului. Evident, formulele au fost stabilite pe baza mai multor presupuneri, rezultate din activitatea de pînă acum a programării.

Metoda lui Champenois de măsurare a calității programelor

Dacă în paragraful precedent calitatea programelor a fost analizată din punctul de vedere al erorilor pe care le conțin, în acest paragraf vom studia structura internă a programului, adică modul eficient sau neeficient în care sînt folosite instrucțiunile sale. Pentru aceasta se va introduce termenul de „impuritate" în scopul măsurării calității programului. Se vor prezenta rezultatele unui studiu efectuat de Champenois [27].

Programul este un ansamblu de operatori și operanzi. Fie notațiile: n_1: numărul de operatori distincți; n_2: numărul de operanzi distincți; N_1: numărul total de apariții a tuturor operatorilor; N_2: numărul total de apariții a tuturor operanzilor.

Atunci: $n = n_1 + n_2$ este dimensiunea vocabularului programului; $N = N_1 + N_2$ este lungimea programului.

Numărul de biți necesari pentru a reprezenta în binar n simboli este $\log_2 n$. Deci numărul de biți necesari pentru a implementa un algoritm de lungimea N cu n simboli distincți este $V = N \log_2 n$, V se numește volumul programului.

Atunci cînd programul se traduce în alt limbaj de programare, volumul său se schimbă deoarece se modifică instrucțiunile sale. Forma cea mai scurtă sub care se poate exprima un algoritm necesită un limbaj ce conține numai numele operanzilor de intrare și de ieșire ai algoritmului (fie n_2' numărul lor) și ca operatori numele procedurii și un operator de atribuire ($N_1 = n_1' = 2$). În acest caz volumul minim (potențial) al programului este:

$$V' = (2 + n_2') \log_2(2 + n_2').$$

Acest volum minim este o valoare constantă caracteristică algoritmului.

Raportul $r = V'/V \leqslant 1$ se numeşte *nivelul programului*, iar inversul lui arată de cîte ori volumul programului este mai mare decît volumul său minim. Cu cît r este mai apropiat de 1 cu atît nivelul programului este mai ridicat, adică este mai apropiat de expresia minimală.

Se poate obţine o aproximare a lui r considerînd că este invers proporţional cu numărul de operatori n_1 şi numărul mediu de apariţii ai fiecărui operand (N_2/n_2). Deci

$$r \simeq \frac{2}{n_1} \cdot \frac{n_2}{N_2} = L.$$

Această aproximare a fost testată pe diverse programe şi a condus la o corelare acceptabilă (0,9 pentru 20 de programe).

Produsul constant $LV \simeq V'$ măsoară conţinutul informaţional al programului. Dar cea mai importantă şi promiţătoare aplicaţie a lui este măsurarea impurităţii programului. Prin impuritate se înţelege stilul greoi de programare, dificultatea de a citi programul, repetiţii inutile de operanzi şi operatori etc.

De exemplu, fie secvenţa de instrucţiuni următoare, luată dintr-un program de sortare, ce plasează elementul $A(i)$ la locul lui, ţinînd cont că porţiunea de tablou $A(1), \ldots, A(i-1)$ este deja sortată crescător:

```
      X = A(I)
10    I = I — 1
      IF(I.EQ.0) GO TO 20
      IF(A(I).LE.X) GO TO 20
      A(I + 1) = A(I)
      GO TO 10
20    A(I + 1) = X
```

Numărul operatorilor şi operanzilor, precum şi numărul lor de apariţii este prezentat în tabelul 12.1.

Tabelul 12.1. Operatori şi operanzi cu număr de apariţii într-un program

operator	nr. apariţii	operand	nr. apariţii
=	4	I	8
+	2	1	3
—	1	O	1
IF	2	A	5
GO TO 20	2	X	3
GO TO 10	1		
EQ	1		
LE	1		
( )	7		
sfîrşit instr.	7		
$n_1 = 10$	$N_1 = 28$	$n_2 = 5$	$N_2 = 20$

$$L = \frac{2}{10} \cdot \frac{5}{20} = 0{,}050.$$

Considerăm o altă variantă a aceluiaşi algoritm:

```
10  I = I — 1
    IF(I.EQ.0) GO TO 20
    IF(A(I).LE.A(I + 1)) GO TO 20
    X = A(I + 1)
    A(I + 1) = A(I)
    A(I) = X
    GO TO 10
20  CONTINUE
```

Numărul operatorilor şi operanzilor este dat în tabela 12.2.

Tabelul 12.2. Operatori şi operanzi cu număr de apariţii în programul modificat

operator	nr. apariţii	operand	nr. apariţii
=	4	I	9
+	2	1	4
—	1	O	1
IF	2	A	6
GO TO 20	2	X	2
GO TO 10	1		
EQ	1		
LE	1		
( )	8		
sfîrşit instr.	8		
continue	1		
$n_1 = 11$	$N_1 = 31$	$n_2 = 5$	$N_2 = 22$

$$L = \frac{2}{11} \cdot \frac{5}{22} = 0{,}041.$$

Puritatea cea mai bună o are prima versiune a algoritmului: 0,05. Să considerăm două programe care au acelaşi număr de instrucţiuni. De exemplu să calculăm cel mai mare divizor comun a două numere întregi pozitive X şi Y prin două metode: algoritmul lui Euclid şi teorema divizorului (dacă c împarte pe x şi y atunci împarte şi diferenţa lor).

Algoritmul lui Euclid

```
10  Q = X/Y
    R = X — Q * Y
    IF(R.EQ.O) GO TO 20
    X = Y
    Y = R
    GO TO 10
20  CMMDC = Y
```

Teorema divizorului

```
10  IF |(X.EQ.Y) GO TO 30
    IF(X.LT.Y) GO TO 20
    X = X — Y
    GO TO 10
20  Y = Y — X
    GO TO 10
30  CMMDC = X
```

Dacă se calculează L pentru cele două programe se obține [27]: — pentru algoritmul lui Euclid: $L = 0{,}086$; — pentru teorema divizorului: $L = 0{,}050$.

Deci puritatea cea mai bună aparține algoritmului lui Euclid, celălalt algoritm fiind mai puțin clar, cu mai multe instrucțiuni GO TO.

Această metodă este foarte potrivită pentru compararea mai multor versiuni ale aceluiași program în scopul de a alege versiunea cea mai eficientă. Pentru aceasta este necesar să se automatizeze procesul de calcul a numărului de operanzi și operatori din program.

Se poate generaliza calculul lui V și L pentru conceperea programelor: se poate aproxima numărul operatorilor și operanzilor în cazul în care descompunerea programului se face cu structurile de control ale programării structurate. Astfel se poate calcula numărul variabilelor intermediare necesare pentru a obține un program pur. În felul acesta se înlătură redondanțele, ambiguitățile și dificultățile de lectură. Rezultatele de mai sus pun în evidență rolul și incidența cantitativă a instrucțiunilor de control ale programului asupra mărimilor ce caracterizează complexitatea sa.

Calitatea și caracteristicile de calitate

Studiile efectuate în cadrul Institutului Central de Conducere și Informatică [84, 85], în cadrul Academiei Șt. Gheorghiu [86] și în cadrul Academiei de Studii Economice [83] abordează într-o manieră mult mai complexă și completă problematica studiului calității produselor-program.

În sinteză aceste studii grupează caracteristicile de calitate în:

a. caracteristici care conferă *utilitate* produsului-program;

b. caracteristici care conferă *operaționalitate* produsului-program (funcționarea corectă a acestuia în mediul de utilizare prestabilit, fără modificări);

c. caracteristici care conferă *adaptabilitate* produsului-program;

d. caracteristici care conferă *transferabilitate* produsului-program.

În mulțimea caracteristicilor de calitate se pot opera clasificări după diverse criterii dintre care menționăm fazele ciclului de viață al programelor și domeniul de aplicație al produsului-program.

Multe din caracteristicile de calitate sînt în relație de dependență unele cu altele. Altele sînt însă în relații conflictuale (vezi fig. 12.2).

Menționăm și faptul că nu toate caracteristicile au valabilitate generală. De la un caz la altul se poate adopta o anumită ierarhie privind importanța caracteristicilor de calitate.

În figura 12.3 prezentăm caracteristicile de calitate *) corespunzătoare celor patru clase enunțate anterior [84]. Atributele din figura 12.3 exprimă ansamblul de proprietăți ale produselor-program care se combină în seturi de criterii necesare pentru caracteristicile de calitate.

---

*) *O caracteristică de calitate* reprezintă grupe relativ omogene de proprietăți calitative subordonate unei clase de caracteristici:

— *Un atribut* de calitate reprezintă un ansamblu omogen de proprietăți elementare de calitate subordonate unei caracteristici de calitate.

— *Proprietăți elementare de calitate* reprezintă acele proprietăți care pot fi măsurate și verificate.

— În procesul de evaluare a calității, proprietățile calitative devin criterii de evaluare a unui produs-program concret.

Fiecare utilizator își poate alege un set de proprietăți de calitate și în consecință stabili un standard de calitate.

În categoria caracteristicilor cu valabilitate generală menționăm: utilitate, fiabilitate, corectitudine, eficiența utilizării resurselor, mentenabilitate.

	CORECTITUDINE	FIABILITATE	UTILIZABILITATE	EFICIENȚA UTILIZĂRII RESURSELOR DE CALCUL	MENTENABILITATE	UȘURINȚA TESTĂRII	FLEXIBILITATE	PORTABILITATE	REUTILIZABILITATE	INTEROPERABILITATE
CORECTITUDINE										
FIABILITATE	○									
UTILIZABILITATE	○	○								
EFICIENȚA UTILIZĂRII RESURSELOR DE CALCUL		◍								
MENTENABILITATE	○	○	○	◍						
UȘURINȚA TESTĂRII	○	○	○	◍	○					
FLEXIBILITATE	○		○	◍	○	○				
PORTABILITATE				◍	○	○				
REUTILIZABILITATE		◍		◍	○	○	○	○		
INTEROPERABILITATE		◍		◍				○		

○ RELAȚII COMPLEMENTARE

◍ RELAȚII CONFLICTUALE

Fig. 12.2. — Relații între caracteristicile de calitate.

Prezintă, de asemenea, interes și gruparea caracteristicilor de calitate în caracteristici puse în evidență în etapele utilizării cuvenite a produsului program (cum ar fi: fiabilitatea, corectitudinea, eficiența, stabilitatea); caracteristici puse în evidență în etapele actualizării produselor program (cum ar fi: mentenabilitatea, adaptabilitatea, claritatea, modificabilitatea); caracteristici puse în evidență în perioadele de tranzacție (portabilitate, integrabilitate) etc.

Măsurarea calității unui produs program pe coordonatele caracteristicilor prezentate în figura 12.3 este deosebit de completă. Pentru aceasta trebuie să existe standarde corespunzătoare și atașați cuantificatori unitari gradelor de satisfacere a fiecărui criteriu în aprecierea calității (caracteristicile cu valabilitate generală pot constitui și criterii în aprecierea calității).

Fiecare utilizator poate acorda un punctaj care să reprezinte gradul de satisfacere a fiecărui criteriu sau, dacă este cazul, subcriteriu.

Prin normative se pot stabili punctajele maxime ce se pot acorda pentru fiecare criteriu de evaluare a calității.

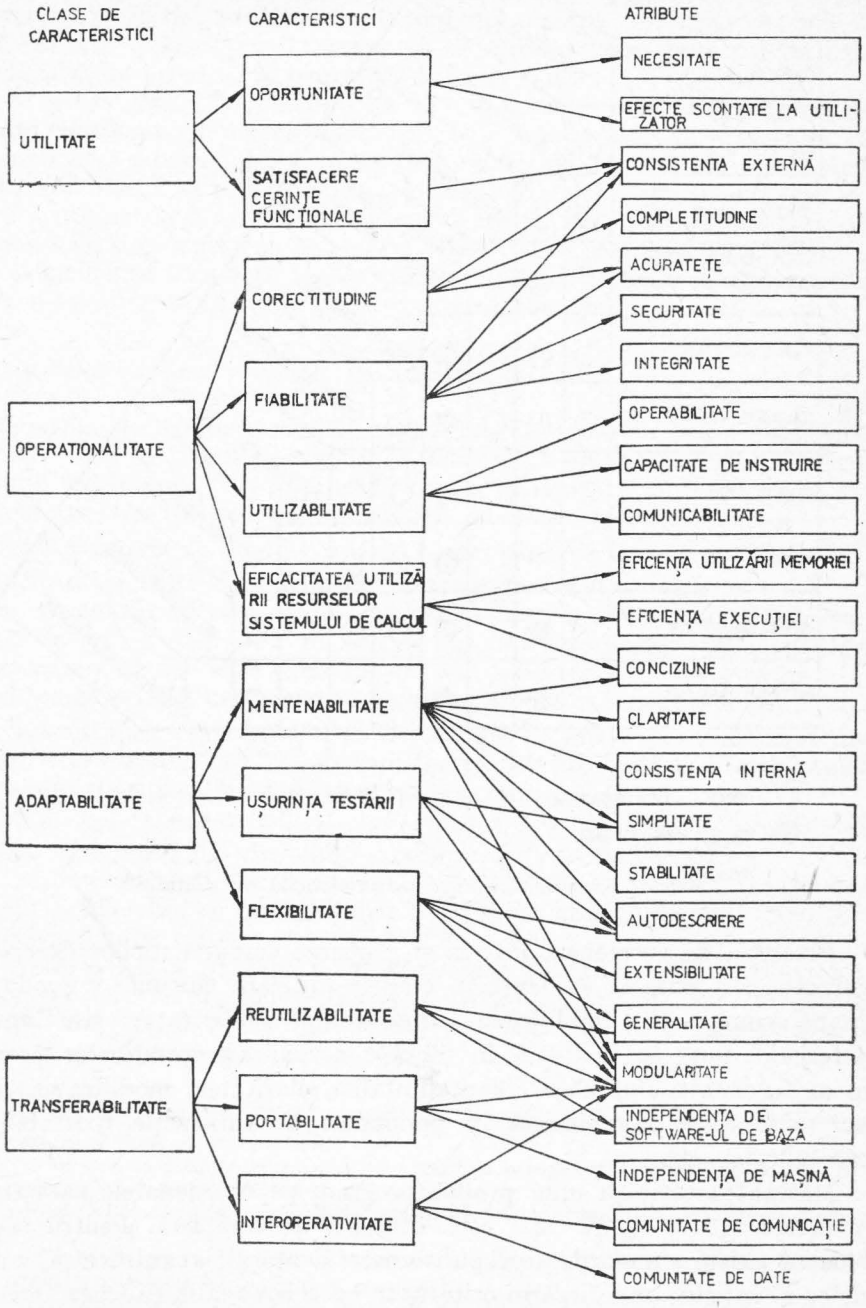


Fig. 12.3. — Relații, clase de caracteristici, caracteristici, atribute.

Formula generală pentru estimarea calității unui program poate fi deci

$$c = \sum_{i=1}^{n} r_i g_i,$$

unde r_i este ponderea acordată cirteriului i, iar g_i gradul de satisfacere a criteriului de calitate i (obținut prin însumarea punctajului acordat fiecărui subcriteriu al criteriului i).

Punerea în relație a lui c cu un c_s (s este standard prestabilit) permite formularea deciziilor cu privire la măsurile ce trebuie luate în vederea creșterii calității produsului program.

Nu trebuie pierdut din vedere însă nici importanța factorului economic în luarea acestor decizii fapt pentru care se impune ca necesară calcularea coeficientului de eficiență [87] folosind formula

$$E = \frac{N \times ef_{\text{med}}}{C_r + N(Ca_{\text{med}} + Ce_{\text{med}})},$$

unde N este numărul de utilizatori potențiali ai produsului program; C_r — cheltuieli de elaborare a produsului program; ef_{med} — efectele economice medii determinate după formula

$$ef_{\text{med}} = \frac{1}{n} \sum_{i=1}^{n} ef_i,$$

unde ef_i sînt efectele economice obținute prin utilizarea produsului program la utilizatorul i;

Ca_{med}, Ce_{med} sînt cheltuieli medii cu adaptarea și cu exploatarea produsului program.

De exemplu:

$$Ca_{\text{med}} = \frac{1}{n} \sum_{i=1}^{n} Ca_i,$$

unde Ca_i sînt cheltuielile utilizatorului i.

Estimarea proprietăților elementare pentru a exprima calitatea produsului-program se poate realiza prin măsurători directe (pe bază de analiză statică sau dinamică a programului) sau prin aprecieri făcute de către specialiști, în ambele cazuri folosindu-se cuantificări în baza unei *scale de măsură* (care poate fi numerică sau logică).

Proprietățile calitative de nivel superior se pot estima pe baza datelor obținute prin estimarea proprietăților elementare și reprezintă ceea ce vom numi *indicatori de calitate*.

Referitor la standardele de calitate facem mențiunea că acestea fac în prezent obiectul a numeroase studii, fără a putea însă vorbi de un sistem de standarde generale riguros fundamentate.

Așa cum se menționează în [85] depistarea unor elemente standardizabile ale procesului de construire a unui standard de calitate particular, pornind de la un metastandard, ar putea constitui o preocupare viitoare.

Problema calității produselor program vizează deopotrivă *programele* și *documentele* corespunzătoare acestora.

Stabilirea unui sistem de indicatori ai calității programelor prezintă un deosebit interes în evaluarea calității și este cerută îndeosebi de cerințele planificării urmăririi și controlului acesteia pe etape ale procesului de elaborare a produselor-program.

Optimizarea calității produselor-program nu poate fi realizată decît în condițiile existenței unui sistem de indicatori de calitate.

Literatura de specialitate citează o serie de indicatori pentru măsurarea calității și îndeosebi pentru evaluarea fiabilității, mentabilității, portabilității, corectitudinii și complexității.

12.2. Metode, tehnici și instrumente de control a calității

Din cele prezentate în paragrafele precedente a rezultat că deși problema calității produselor program este de o maximă importanță încă nu putem vorbi de un sistem general de indicatori și nici de norme și standarde de apreciere a acesteia.

În aceste condiții nu putem vorbi încă de o singură metodă de măsurare a calității.

În acest paragraf vom prezenta într-o formă adaptată din [85] problematica metodelor și tehnicilor de control a calității.

Reținem mai întîi gruparea acestor metode și tehnici în:

— metode și tehnici orientate strict spre evaluare, adică pe măsurarea și compararea cu un etalon;

— metode și tehnici orientate și pe aspectele organizatorice ale activității de control.

Prima categorie de metode presupune folosirea în exclusivitate a unui sistem agregat de indicatori de calitate ce măsoară nivelul fiecărei caracteristici, fiecare grupă de caracteristici și nivelul calitativ al produsului program în ansamblu. Ea presupune și o anumită scală de măsurare a proprietăților calitative ale produsului program.

În sistemul de indicatori, construit în formă piramidală, se includ trei categorii de indicatori (de la bază spre vîrful piramidei);

— *indicatori analitici:* cuantifică fiecare atribut al produsului program;

— *indicatori sintetici:* cuantifică fiecare caracteristică a produsului program;

— *indicatori globali:* cuantifică calitatea produsului program în ansamblu.

În această primă categorie sînt incluse:

— *metodele numerice* — au scală de măsurare numerică;

— *metode nenumerice (logice)* — au scală de măsurare nenumerică iar indicatorii de calitate reprezentînd scheme de raționament logic.

Fiecare metodă se particularizează prin anumite tehnici de evaluare care precizează maniera de calcul a indicatorilor de calitate.

Vom delimita deci *tehnici axate strict pe evaluare și tehnici care au în vedere și aspectele organizatorice ale activității de control.*

În ceea ce privește tehnicile axate strict pe evaluare putem spune că ele nu au fost încă elaborate. Realizările de pînă acum se limitează în general la indicatori simpli și care exprimă doar parțial calitatea. Cităm ca titlu de exemplu matricele Halstead pentru măsurarea algoritmelor (codului sursă) propuse pentru evaluarea calității produselor program.

Tehnicile care au în vedere și aspectele organizatorice ale activității de control vizează nu atît măsurarea nivelului de calitate al produselor program cît mai ales depistarea erorilor și problemelor de proiectare/programare. Menționăm din această categorie tehnici cum ar fi *inspecția de cod* și *verificările structurate* [88].

Inspecția de cod este o tehnică manuală care presupune verificarea codului sursă sau a specificațiilor de programare pe diferite niveluri ale procesului de elaborare a programelor pentru a depista erorile sintactice și semantice.

Verificările structurate reprezintă o tehnică bazată pe simularea manuală a funcționării programului și se poate aplica în toate etapele procesului de elaborare a programelor.

Unei tehnici de calcul îi sînt specifice maniera de selecție și măsurare a proprietăților elementare și concretizarea modelului general al indicatorilor de calitate.

Tehnicile pot fi clasice, adică folosite curent la măsurarea calității produselor industriale, sau *specifice* măsurării produselor-program.

Desigur că este de dorit ca pe baza studierii tehnicilor clasice și a specificațiilor produselor-program să se ajungă la tehnici care să reflecte cît mai fidel aceste specificații.

În concepția măsurării calității pe etape de ciclu de viață al programelor vor exista tehnici specifice acestor etape.

În cadrul metodei se precizează și maniera de validare a indicatorilor de calitate. Spre exemplu, prin corelare statistică se pot valida indicatorii obținuți printr-o tehnică ce se încadrează într-o metodă numerică de evaluare.

Instrumentele folosite în controlul calității produselor program sînt în general incluse în categoria instrumentelor de testare prin care se urmărește verificarea semantică a produsului obținut în fiecare etapă a produsului de elaborare a acestora.

Ulterior, odată cu extinderea metodelor evoluate de programare și organizare a acestei activități s-a făcut distincția între instrumente de testare a produsului-program și instrumente de testare a proiectului acestuia, acestea din urmă numite deseori și „instrumente de verificare și validare".

Din prima categorie de instrumente fac parte: instrumente de analiză statică a codului sursă (fără a necesita execuția programului)și instrumente de testare dinamică a programelor (vezi capitolul 8).

Există instrumente care automatizează doar calculul indicatorilor de calitate și instrumente care automatizează inclusiv cuantificarea proprietăților elementare ale produselor-program (memorate de obicei într-o bază de date).

În funcție de gradul de integrare în tehnologia de elaborare a produselor-program distingem *instrumente de control a calității înglobate* într-un mediu de elaborare a produselor-program și *instrumente de control a calității independente*, utilizate în fazele procesului de elaborare al programelor.

Sistemul PACSIN dezvoltat în cadrul I.C.I. cuprinde și instrumente de control a calității programelor.

Studiile de elaborare a programelor AIDES [89], ARGUS [90], SEF [91] etc. dezvoltate pe plan mondial conțin, de asemenea, instrumente de control a calității programelor.

Tendința pe plan mondial și la noi în țară este de a construi instrumente integrate de evaluare a calității produselor-program și, de asemenea, de evaluare a calității procesului de elaborare a acestora.

Metoda de programare și tehnologia cadru de elaborare a programelor trebuie să aibă în vedere și aspectele legate de efectuarea controlului de calitate.

12.3. Un mecanism pentru îmbunătățirea calității programelor

Calitatea programelor depinde de aplicarea noilor tehnici de elaborare a programelor: concepere descendentă, programare structurată, organizarea corespunzătoare a datelor etc. Evident, ea depinde de stilul și experiența programatorului. Există însă și mecanisme hardware pentru detectare și eliminarea cît mai rapidă a erorilor. Un astfel de mecanism este cel cunoscut sub numele de ARAR (Assign-Read-Assing-Read) [28], ce se bazează pe următoarea idee: o variabilă care a primit o valoare trebuie mai întîi citită înainte de a primi o nouă valoare, altfel se consideră eroare. Deci ciclul de viață al unei variabile este: alocarea spațiului; atribuirea unei valori; citirea valorii variabilei, eventual de mai multe ori; eliberarea spațiului ocupat de variabilă.

Fie exemplul de mai jos:

```
begin
    integer a, b, c
        a = 5
        c = 4
    . . . . . . . . . . . . . . . . .
    instrucțiuni ce nu utilizează variabila a
    . . . . . . . . . . . . . . . . .
    a = b + c
    . . . . . . . . . . . . . . . . .
end
```

Dacă programul este corect, atunci prima instrucțiune (a = 5) este de prisos pentru că mai jos (a = b+c) se asignează o nouă valoare fără a se folosi prima. Deci instrucțiunea a = 5 trebuie eliminată, obținîndu-se un program mai scurt și mai rapid. Dacă programul este eronat și în loc de a = 5 trebuie să fie b = 5, atunci, un control de tip ARAR ar fi detectat eroarea.

Pe de altă parte există cazuri în care se fac asignări de valori unei variabile în mod succesiv, fără ca programul să fie eronat. Fie de exemplu secvența următoare, scrisă în ALGOL W în care elementele diagonale ale unei matrice se inițializează cu alte valori:

```
for i = 1 until n do;
begin
    for j = 1 until n do; a(i, j) = 1;
    a(i, i) = 0;
end
```

Pentru a respecta principiul controlului ARAR programul ar trebui scris astfel:

```
for i = 1 until n do;
for j = 1 until n do;
    a(i, j) = if j = i then 0 else 1;
```

Nu întotdeauna însă se pot face astfel de modificări. Din acest motiv se introduce o nouă instrucțiune prin care se anulează valoarea variabilei (se spune că se eliberează variabila). De exemplu:

```
free a
free a, b, c, x(4)
```

Cu această instrucțiune exemplul precedent se scrie astfel:

```
for i = 1 until n do;
begin
    for j = 1 until n do; a(i, j) = 1;
    free a(i, i);
    a(i, i) = 0;
end
```

Implementarea harware a controlului ARAR se poate face astfel: la fiecare cuvînt din memorie se asociază un bit suplimentar b_0 care ia următoarele valori: $b_0 = 1$, valoarea atribuită cuvîntului nu a fost citită; $b_0 = 0$ — valoarea atribuită cuvîntului a fost citită.

Dacă se încearcă să se stocheze o valoare într-un cuvînt al cărui bit $b_0 = 1$ se produce o derutare. Inițial $b_0 = 0$ pentru toate locațiile programului. Instrucțiunea hardware de stocare a unui registru r într-o locație w de memorie se modifică astfel:

```
dacă b0(w) = 1 atunci
    eroare
altfel
    w: = r
    b0(w) = 1
```

Implementarea software a principiului ARAR a fost realizată pentru elaborarea compilatorului BABEL [29]. Bitul b_0 se reprezintă pe un cuvînt, ceea ce face să se dubleze zona de date a programului și să scadă viteza de execuție. Avantajul constă în detectarea unor erori greu de descoperit prin alte mijloace, îmbunătățindu-se simțitor calitatea programelor.

12.4. Optimizarea programelor

Problema optimizării programelor este legată de cea a calității lor. Tehnicile prin care se îmbunătățește calitatea programelor optimizează uneori anumite funcții ale acestuia, și invers, prin optimizarea unui program crește de fapt calitatea lui.

Termenul de optimizare este totuși impropriu, deoarece în general nu se poate stabili o funcție de optimabilitate prin care să se precizeze nivelul de performanță al programului. De asemenea, nu se poate defini mulțimea tuturor programelor destinate rezolvării unei probleme din care să se aleagă programul cel mai performant. De aceea, optimizarea programului se referă la ameliorarea funcțiilor lui în raport cu anumite criterii ce depind de natura utilizării programului. Astfel putem dori un program: ușor de modificat; modular; adaptabil la noi condiții de exploatare; fiabil; să nu ocupe mult spațiu de memorie; rapid etc.

În momentul în care se precizează cadrul utilizării programului, optimalitatea se definește în raport cu acest cadru. De obicei cele mai multe criterii de optimizare se bazează pe: timpul de execuție; spațiul de memorie ocupat.

Se știe că problema generală de a găsi un program optim este indecidabilă. De aceea tehnicile aplicate programelor pentru ameliorarea funcțiilor lor țin de domeniul experienței și intuiției programatorului.

Multe din tehnicile de optimizare a programelor au repercusiuni asupra clarității acestora. Experiența de pînă acum a activității de programare arată că de cele mai multe ori claritatea programului este mai importantă decît eficiența lui. Programele clare sînt mai ușor de modificat, actualizat, revizuit etc. chiar de alte persoane decît cele care le-au elaborat. De foarte multe ori este mai avantajos de a avea un program adaptabil, ușor de întreținut, decît un program eficient dar înghețat.

De aceea tehnicile de optimizare a programelor se aplică numai atunci cînd eficiența programului se impune în mod necesar: de exemplu, programul nu intră în partiție sau nu asigură un timp de răspuns suficient de rapid la anumite evenimente. Așa cum s-a mai spus eficiența sau performanța programului se referă în general la modul de execuție al programului, și anume la: timpul de execuție; memoria ocupată.

De fapt acești doi factori constituie și elementele de bază pentru contabilizarea utilizării resurselor de către un program. Tehnicile de optimizare prezentate în continuare se referă la reducerea timpului de execuție și spațiului de memorie necesar programului. Practic optimizarea programelor se realizează în decursul următoarelor etape: conceperea programului; codificarea programului; compilarea programului.

12.4.1. Optimizarea programelor în etapa de concepere

În etapa de concepere a programului se pune problema alegerii algoritmului ce rezolvă problema cu un număr redus de operații. Nu există o metodă de determinare a unui asemenea algoritm; totul depinde de experiența și intuiția programatorului.

Fie de exemplu problema următoare, propusă de Dijkstra, cunoscută și sub numele de drapelul olandez: se dă un șir de pioni numerotați de la 1 la n, fiecare avînd o culoare: albastru, alb sau roșu. Pionii fiind într-o ordine oarecare se cere să se reordoneze șirul în așa fel încît cei de culoare albastră să fie la început, cei de culoare albă — la mijloc și cei roșii la sfîrșit. Singura operație permisă este schimbarea locului a doi pioni. De asemenea, pentru a afla culoarea pionului trebuie făcut un singur acces la el.

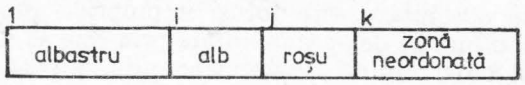


Fig. 12.4. — Rezolvare parțială.

Rezolvare. Problema devine foarte simplă dacă se presupune că este parțial rezolvată (fig. 12.4), adică o parte din pioni sînt deja grupați în cele 3 zone iar o altă parte nu este încă explorată.

Se continuă rezolvarea problemei prin prelucrarea primului pion din zona neordonată. Se face acces la pionul K și se testează culoarea lui: dacă este roșu nu se face nimic și se avansează la $K + 1$; dacă este alb se schimbă cu primul pion roșu (j) și se avansează la $K + 1$; dacă este albastru se schimbă cu primul pion alb (i) ajungîndu-se la situația precedentă.

Pentru pionul $K + 1$ se face același prelucrare ca mai sus. Se continuă astfel pînă se plasează și ultimul pion n. Inițial cele trei zone sînt vide. Algoritmul se programează în pseudocod astfel:

```
i: = 1, j: = 1, k: = 1
cît timp k ≤ n execută
        c: = culoare p(k)
        dacă c = roșu
        atunci k: = k + 1
        altfel dacă c = alb
                atunci
                        schimbă p(j) cu p(k)
                        j: = j + 1
                        k: = k + 1
                altfel schimbă p(i) cu p(k)
                        i: = i + 1
                        schimbă p(j) cu p(k)
                        j: = j + 1
                        k: = k + 1
                sfîrșit
        sfîrșit
sfîrșit ciclu
```

Să presupunem acum că zona neordonată se află între zona de pioni albaștri și albi și se păstrează aceleași notații ca în figura precedentă (fig. 12.5).

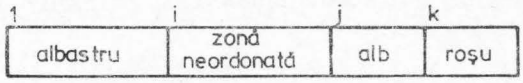


Fig. 12.5. — Ipoteza de lucru cu integrarea zonei neordonate între pionii albastru și alb.

În acest caz se obține următorul program:

```
i: = 1, j: = n + 1, k: = n + 1
pînă ce i = j execută
        c: = culoare (j − 1)
        dacă c = alb
        atunci j: = j − 1
        altfel dacă c = albastru
                atunci
                        schimbă p(i) cu p(j − 1)
                        i: = i + 1
                altfel j: = j − 1
                        k: = k − 1
                        schimbă p(j) cu p(k)
                sfîrșit
        sfîrșit
sfîrșit ciclu
```

Se observă că algoritmul al doilea realizează mai puține operații de mutare a pionilor. Deci pentru o problemă simplă s-au construit două programe diferite ca viteză de calcul și aceasta numai pe baza unor ipoteze de recurență. Nu există însă nici o metodă de a transforma primul program pentru a-l obține pe al doilea. De aici rezultă importanța deosebită a fazei de concepere a programului, deoarece de această fază depinde calitatea lui.

Problema care se pune la optimizarea unui program este de a ști ce trebuie optimizat. Numeroși cercetători sînt convinși că scrierea unui program de calitate nu se face dintr-o dată. În general se procedează astfel: se scrie un program valid, indiferent de calitate; se ameliorează programul precedent în diverse direcții: simplitate, claritate, rapiditate, lungime text etc.

Acest procedeu prezintă interes numai dacă modificările făcute în program sînt minore sau bine localizate. Altfel există riscul să se introducă o mulțime de erori care pot face programul inutilizabil. În cazul programelor complexe se localizează acele porțiuni critice (prin simulări, date de test etc.) de care depind performanțele programului și se modifică în sensul optimizării lor. De exemplu, secvența de instrucțiuni următoare plasează elementul $a(i)$ conform valorii lui, într-o secvență $a(1), \ldots, a(i-1)$, deja ordonată crescător:

```
k: = i — 1
cît timp k > 0 execută
        dacă a(k) ⩽ a(k + 1) atunci ieșire din ciclu;
        x: = a(k + 1); a(k + 1): = a(k); a(k): = x;
        k: = k — 1
sfîrșit ciclu
```

Dacă se analizează acest program se observă că $a(i)$ se schimbă cu toți predecesorii lui. Programul devine mai rapid dacă se păstrează $a(i)$ în variabila x și se decalează de un loc spre dreapta toate elementele mai mari ca x, după care se depune x în locul rămas liber. Secvența de instrucțiuni de mai sus se transformă astfel:

```
k: = i — 1;
x: = a(k);
cît timp k > 0 execută
      dacă a(i) ⩽ x atunci ieșire din ciclu;
      a(k + 1): = a(k);
      k: = k — 1;
sfîrșit ciclu
a(k + 1): = x.
```

12.4.2. Optimizarea programelor în etapa de codificare

La codificarea programului trebuie să se țină cont de redactarea expresiilor de calcul într-o formă cît mai eficientă din punct de vedere al timpului de execuție și al memoriei ocupate. Astfel trebuie eliminate pe cît posibil instrucțiunile de calcul redondante sau inutile.

În primul rînd trebuie să se aleagă limbajul de programare adecvat preluării executate de algoritm. Un limbaj necorespunzător presupune introducerea de noi secvențe de calcul pentru suplinirea structurilor de control inexistente.

Pentru codificarea programului în limbajul de programare se fac următoarele recomandări:

a) Să se inițializeze variabilele la compilare (dacă permite limbajul de programare).

Exemplu (limbaj PL/1)

```
DECLARE (A,B) INTEGER INITIAL (100,500)
```

În acest caz prin inițializarea variabilelor la compilare se elimină instrucțiunile de atribuire de valori inițiale din cadrul programului și anume:

```
A = 100;
B = 500.
```

b) Să se evite pe cît posibil conversiile de formate de variabile.

Fie, de exemplu, instrucțiunile următoare scrise în limbajul FORTRAN:

```
X = 0
Y = 5/4.
```

Dacă x și y sînt variabile de tip real atunci se va face apel la rutina de conversie a valorilor de atribuire întregi în valori reale.

Conversiile se evită dacă instrucțiunile se scriu astfel:

```
X = 0.
Y = 5./4.
```

De asemenea, secvența următoare scrisă în limbajul COBOL

```
77 X PIC S9V9
77 Y PIC S99V99
.....................
ADD X TO Y
```

presupune transformări interne pentru alinierea punctului zecimal al celor două variabile. Aceste transformări se pot evita dacă se declară variabila x astfel:

```
77 X PIC S99V99
```

Trebuie evitate pe cît posibil conversiile de variabile în cadrul expresiilor de calcul. Fie de exemplu secvența de instrucțiuni următoare scrisă în limbaj FORTRAN:

```
X = Y/I
Z = (Y + I)/4.0
V = (Y + Y)/I
```

Dacă x, y, z și v sînt variabile de tip real, iar I este de tip întreg, atunci la execuția fiecărei instrucțiuni de mai sus trebuie realizată o conversie întreg-real. Numărul de conversii se reduce la 1 dacă se scrie secvența de instrucțiune astfel:

```
XI = I
X = Y/XI
Z = (Y + XI)/4.0
V = (Y + Y)/XI
```

c) Să se utilizeze cu precădere operațiile aritmetice rapide. În funcție de viteza de execuție operațiile aritmetice se ordonează astfel: adunarea și scăderea (+, —); înmulțirea (*); împărțirea (/); ridicarea la putere (**).

Calculul expresiilor aritmetice este rapid dacă se folosesc operații aritmetice rapide. De aceea atunci cînd este posibil se înlocuiesc operațiile mai puțin rapide cu cele rapide. De exemplu:

2 * I se înlocuiește cu I + I
I ** 2 se înlocuiește cu I * I
X/5. se înlocuiește cu X * 0.2
X** 5. se înlocuiește cu X**5 deoarece ridicarea la o putere întreagă este mai rapidă decît ridicarea la o putere reală.

d) Să se elimine subexpresiile comune.

Dacă o subexpresie există în mai multe expresii de calcul atunci se notează cu o variabilă pentru a evita calcularea repetată a ei în toate expresiile în care apare. De exemplu, fie secvența de instrucțiuni următoare:

```
A = (X + Y) ** 5 + B
B = (X + Y) ** 5 + X
C = (X + Y) ** 5 + Y
```

Ea se poate optimiza, din punct de vedere al timpului de execuție astfel:

```
Z = (X + Y) ** 5
A = Z + B
B = Z + X
C = Z + Y
```

Uneori este necesar de a rearanja expresiile pentru a pune în evidență subexpresiile comune. De exemplu, fie secvența de instrucțiuni următoare:

```
X = X + A * 2 — B + Y
Z = Z — Y — 2 * A + B
```

Dacă se aranjează expresia lui Z astfel:

```
Z = Z — (A * 2 — B + Y)
```

atunci secvența de instrucțiuni se poate optimiza ca mai jos:

$$V = A * 2 - B + Y$$
$$X = X + V$$
$$Z = Z - V$$

e) Să se optimizeze secvențele repetitive.

În primul rînd trebuie eliminate expresiile invariante din cadrul ciclurilor. Fie, de exemplu, secvența repetitivă de mai jos scrisă în FORTRAN:

```
      DØ 10 I = 1, 100
         A(I) = B * C * D + X(I)
   10 CØNTINUE
```

Calculul repetat al subexpresiei B * C * D se elimină astfel:

```
      Z = B * C * D
      DØ 10 I = 1, 100
         A(I) = Z + X(I)
   10 CØNTINUE
```

De asemenea, pentru a evita repetarea calculelor de intrare într-un ciclu și de ieșire din ciclu se aplică tehnica grupării ciclurilor cu același număr de iterații. Fie secvențele repetitive de mai jos:

```
      DØ 10 I = 1, 1000
         A(I) = 0.
   10 CØNTINUE
      DØ 20 I = 1, 1000
         B(I) = 1.
   20 CØNTINUE
```

Cele două cicluri pot fi grupate împreună:

```
      DØ 10 I = 1, 1000
         A(I) = 0.
         B(I) = 1.
   10 CØNTINUE
```

De asemenea, pentru a minimiza numărul de operații de intrare în ciclu și de ieșire din ciclu se recomandă ca ciclurile independente imbricate să fie grupate în ordinea crescătoare a numărului de iterații. În exemplul următor numărul de operații de intrare și ieșire din ciclu este maxim.

```
DØ 100 I = 1,30                1 operație de intrare în ciclu
      DØ 50 I = 1,20          30 operații de intrare în ciclu
            DØ 25 K = 1,10   600 operații de intrare în ciclu
.....................
25          CØNTINUE        6000 operații de ieșire din ciclu
50     CØNTINUE              600 operații de ieșire din ciclu
100 CØNTINUE                  30 operații de ieșire din ciclu
```

Dacă organizarea ciclurilor prezentată mai jos este posibilă numărul operațiilor de intrare și ieșire din ciclu este minim:

```
DØ 100 K = 1,10          1 operație de intrare în ciclu
   DØ 50 J = 1,20        10 operații de intrare în ciclu
      DØ 25 I = 1,30     200 operații de intrare în ciclu
.....................
25        CØNTINUE       6000 operații de ieșire din ciclu
50     CØNTINUE          200 operații de ieșire din ciclu
100 CØNTINUE             10 operații de ieșire din ciclu
```

Prin artificiile de programare prezentate mai sus se urmărește reducerea timpului de execuție al programului. Cele prezentate nu epuizează problema în cauză. Astfel viteza de execuție a programului mai depinde și de operațiile de intrare-ieșire pe care le execută, precum și de organizarea datelor sale.

În ceea ce privește operațiile de intrare-ieșire se recomandă reducerea lor pe cît posibil sau utilizarea unor dispozitive periferice rapide pentru stocarea și citirea datelor. De asemenea, programatorul trebuie să cunoască în detaliu tehnicile de organizare a fișierelor și modul de acces la ele pentru a alege organizarea cea mai potrivită aplicației sale.

În ceea ce privește structura datelor, trebuie să se aleagă acele structuri care minimizează numărul de operații pentru stocarea sau regăsirea datelor. De exemplu, organizarea secvențială de tablou permite un timp de acces mai rapid decît parcurgerea unei liste. La adresarea elementelor unui tablou viteza de acces crește dacă se evită calcule repetate asupra indecșilor.

Fie secvența repetitivă următoare:

```
DØ 10 I = 1, N
   A(3 * I + 2) = B(3 * I + 2) + C(3 * I + 2) + X
10 CØNTINUE
```

Un mod eficient de realizarea este:

```
DØ 10 I = 1,N
   K = 3 * I + 2
   A(K) = B(K) + C(K) + X
10 CØNTINUE
```

Instrucțiunile EQUIVALENCE, pe care le posedă unele limbaje, permit obținerea unui cîștig important la execuția programului, dar se pierde din claritate. De exemplu:

```
DØ 20 I = 1,N
   DØ 10 J = 1,M
      A(I, J) = 0.
10    CØNTINUE
20    CØNTINUE
```

Se introduce un nou tablou B, de o singură dimensiune, cu N * M elemente, echivalente cu tabloul A:

```
      EQUIVALENCE (A(1, 1), B(1))
      NM = N * M
      DØ 10 I = 1,NM
        B(I) = 0.
   10 CØNTINUE
```

Problema reducerii spațiului de memorie ocupat de program este importantă din următoarele puncte de vedere:

— aplicațiile utilizatorului au devenit tot mai complexe și există riscul ca memoria disponibilă a calculatorului să nu fie suficientă;

— odată cu introducerea conceptului de multiprogramare spațiul de memorie pus la dispoziția unui program a devenit mai mic prin împărțirea memoriei în partiții.

Reducerea dimensiunilor unui program este o problemă dificilă. O bună programare reduce codul executabil, dar spațiul ocupat de date (tablouri, liste) nu poate fi micșorat. De aceea, în cazul în care dimensiunea programului devine critică se memorează unele structuri de date pe suporturi externe (disc), în dauna vitezei de execuție a programului.

O tehnică devenită deja clasică pentru reducerea dimensiunilor programului este segmentarea. Programul se împarte în mai multe părți relativ independente, numite segmente, iar încărcarea lor în memorie se face numai cînd este nevoie. Același spațiu de memorie va fi utilizat succesiv pentru execuția mai multor segmente. Structura de acoperire a segmentelor în memorie se reprezintă printr-o arborescență. Segmentele unui program care pot fi simultan în memorie se află pe o ramură a structurii de arborescență. De exemplu, în figura 12.6 dimensiunea maximă a programului este dată de prezența simultană în memorie a segmentelor A, C, E.

Ramurile au o parte comună, numită rădăcină. Ea trebuie să se afle permanent în memorie și conține cel puțin instrucțiunile de apel ale segmentelor imediate următoare de pe toate ramurile precum și datele comune de lucru pentru segmentele de pe ramuri diferite.

Încărcarea unui segment în memorie se poate face: explicit prin instrucțiune de încărcare segment; implicit prin apelul unei subrutine din interiorul segmentului.

De fiecare dată cînd se aduce un segment în memorie nu se salvează datele obținute de la încărcarea precedentă. De aceea datele cu valori permanente trebuie să se afle în rădăcină.

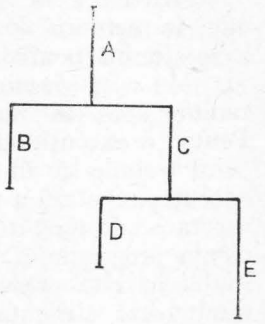


Fig. 12.6. — Structura pe segmente a unui program.

Descrierea structurii de acoperire a programului se face la editarea de legături a acestuia. Pentru diverse sisteme există diverse formate pentru această descriere. De exemplu, în sistemul SIRIS de pe calculatoarele FELIX, structura arborescentă din figura 12.6 se descrie printr-o cartelă TREE ce se transmite editorului de legătură:

```
. TREE (A(B, C(D, E)))
```

Prin segmentarea programului scade eficiența sa ca timp de execuție. De aceea se recomandă să se evite împărțirea programului în segmente mici,

deci numeroase: pe lîngă faptul că traficul cu discul crește simțitor se face și consum de memorie pentru descrierea fiecărui segment.

O altă tehnică de economisire de memorie, ce se aplică în sistemele cu partiționarea memoriei pentru multiprogramare, se referă la construirea și utilizarea de blocuri comune de date sau partajarea de biblioteci comune.

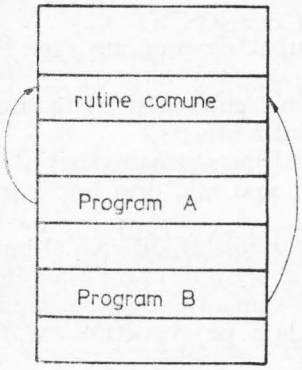


Fig. 12.7. — Programe cu partiții comune.

În multe sisteme de operare se pot declara partiții comune (la care au acces mai multe programe) ce conțin colecții de rutine, numite biblioteci partajate, sau zone de date comune (fig. 12.7).

În cazul unei aplicații complexe formată din mai multe programe ce se execută simultan, utilizatorul poate grupa toate rutinele comune într-o singură partiție comună, realizîndu-se pe ansamblu un cîștig important de memorie. Programele se leagă la rutinele din bibliotecile partajate la editarea de legături.

În bibliotecile partajate se pot introduce și rutinele ce execută anumite funcții sistem (de exemplu rutinele SGF de acces la fișiere), care în mod normal sînt incluse automat în fiecare program la editarea de legături.

Rutinele din bibliotecile partajate trebuie scrise reentrant, iar accesul la ele trebuie să se facă numai în citire.

În sistemele cu memorie virtuală, în care memoria fizică este paginată nu se mai pune problema spațiului de memorie ocupat de program. În aceste sisteme utilizatorul are la dispoziția sa o memorie virtuală de mari dimensiuni ce se proiectează sub controlul sistemului de operare în spațiul fizic. Memoria fizică este împărțită în segmente (pagini) de lungime fixă, în care se încarcă segmentele (paginile) din spațiul virtual al programului.

Eficiența în execuție a programului scade: paginile programului se aduc în memorie de pe suportul extern de memorare, atunci cînd este nevoie și se elimină pentru a face loc la alte pagini.

În acest caz operațiile cu suportul de memorare al programului (disc, tambur etc.) sînt numeroase, ceea ce face să crească timpul său de execuție. Pentru o execuție eficientă în contextul utilizării memoriei paginate, utilizatorul trebuie să aibă în vedere noțiunea de localitate a programului, adică execuția repetată a unui grup de instrucțiuni într-un spațiu restrîns. În felul acesta scade numărul de pagini ce trebuie aduse în memorie deci crește eficiența programului. Dimpotrivă, salturile numeroase dintr-o zonă a programului în alta presupune încărcări repetate de pagini în memorie, ceea ce diminuează eficiența programului. De asemenea este important ca programatorul să grupeze subrutinele în spațiul programului în ordinea lor de apel. Cu cît este mai mare gradul de localitate al programului, cu atît execuția sa este mai rapidă.

12.4.3. Optimizarea programelor în etapa de compilare

Există compilatoare (ca PL/1 de exemplu) la care se pot specifica opțiuni de optimizare a codului generat. Astfel utilizatorul poate cere minimizarea timpului de execuție al programului sau reducerea spațiului de memorie ocupat.

Optimizarea în ceea ce privește timpul de execuție se referă la eliminarea de subexpresii comune. În acest caz programatorul trebuie să rearanjeze subexpresiile comune în așa fel încît să poată fi recunoscute ușor de compilator. De exemplu:

$$X = X + A * B + A - B$$

$$Y = Y + B * A - A + B$$

În general compilatorul nu recunoaște că A * B este identic cu B * A. De aceea expresia lui Y trebuie rescrisă ca:

$$Y = Y + A * B - (A - B)$$

Compilatorul încearcă să optimizeze secvențele de instrucțiuni din cadrul unui bloc (delimitat de instrucțiunile BEGIN, END). Tot pentru o execuție eficientă a programului, compilatorul atribuie cîte un registru din unitatea centrală pentru fiecare variabilă frecvent utilizată din cadrul unui bloc. Dacă programatorul aplică tehnicile avansate de programare, programul va rezulta ca o succesiune de blocuri de acțiuni, în care salturile explicite (GØTØ) sînt reduse sau inexistente. În acest caz procesul de optimizare aplicat de compilator este mult ușurat, fiecare bloc fiind o secvență de instrucțiuni de optimizat.

12.4.4. **Transformarea programelor recursive**

O problemă mult discutată în literatura de specialitate privind optimizarea este transformarea programelor recursive în programe iterative. Procedeul recursivității este foarte des utilizat pentru că permite o descriere simplă a unor algoritmi. Uneori este costisitor, adică programele recursive sînt mai puțin rapide. Pe de altă parte anumite limbaje utilizate frecvent (ca FORTRAN de exemplu) nu permit codificarea recursivă a algoritmilor. Problema transformării programelor recursive în programe iterative este formalizată și în principiu constă în următoarele: (a) eliminarea parametrilor procedurii recursive; (b) transformarea apelurilor recursive în iterații.

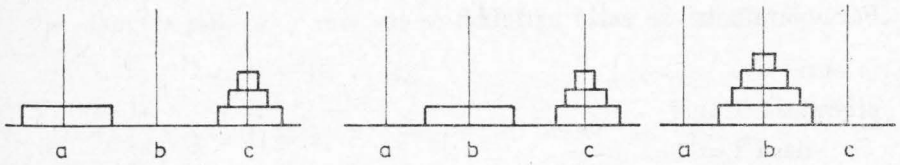


Fig. 12.8. — Problema turnurilor din Hanoi: model 1.

Prezentăm procedura de transformare a programului recursiv în program iterativ așa cum apare în [30], oprindu-se numai la suprimarea apelurilor recursive (punctul b). Se consideră problema turnurilor din Hanoi, care are enunțul următor: fiind date 3 suporturi *a*, *b*, *c*, pe care se pot introduce *n* discuri de diametre diferite, se definește un turn ca un șir vertical de discuri de diametre descrescătoare. Se cere să se transporte un turn de pe suportul *a* pe suportul *b*, deplasînd discurile unul cîte unul, singurul stocaj permis fiind în diametre descrescătoare. Suportul *c* este ajutător (fig. 12.8).

Pentru un singur disc problema este simplă:

Hanoi (a, b, c, 1) = deplasare (a, b)

Să presupunem că se știe să se rezolve problema pentru $n - 1$ discuri. Pentru a rezolva Hanoi (a, b, c, n) este suficient de a face:

Hanoi (a, c, b, n — 1); Hanoi (a, b, c, 1,); Hanoi (c, b, a, n — 1) așa cum se arată în figura 12.9.

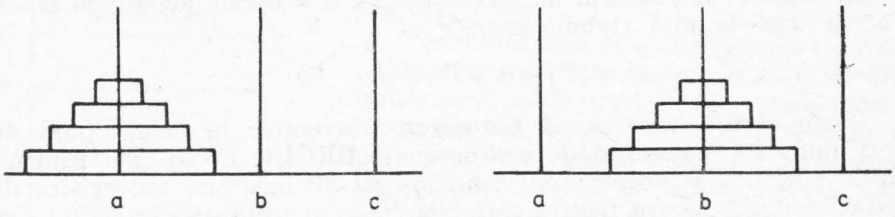


Fig. 12.9. — Problema turnurilor din Hanoi: model 2.

Problema se rezolvă prin procedura următoare:

```
i: = n
procedură Hanoi (a, b, c, i);
    dacă i = 1
    atunci deplasează (a, b);
    altfel
        Hanoi (a, c, b, i—1);
        deplasează (a, b);
        Hanoi (c, b, a, i—1)
    sfîrșit
Sfîrșit procedură
```

Fără parametri se scrie astfel:

```
i: = n
procedură Hanoi;
    dacă i = 1
    atunci deplasează (a, b);
    altfel i: = i — 1; schimbare (b, c); Hanoi; i: = i + 1;
                schimbare (b, c);
            deplasare (a, b);
            i: = i — 1; schimbare (a, c); Hanoi;
                i: = i + 1; schimbare (a, c)
    sfîrșit
sfîrșit procedură
```

Vrem să transformăm procedura recursivă de mai sus în procedură iterativă. Se poate pune sub forma următoare:

```
i: = n
procedură Hanoi;
    dacă i = 1
    atunci x;
    altfel y;
        Hanoi;
        z;
        Hanoi;
        t;
    sfîrşit
sfîrşit procedură
```

unde:

x:: = <deplasare (a, b)>

y:: = <i : = i — 1; schimbare (b, c)>

z:: = <schimbare (b, c); deplasare (a, b); schimbare (a, c)>

t:: = < i: = i + 1; schimbare (a, c)>

La definirea lui z s-au eliminat instrucțiunile i: = i + 1 și i: = i — 1 fiind inefective.

Folosim forma axiomatică de reprezentare a instrucțiunii datorată lui Hoare [43]:

$$P\{S\}R$$

unde P este predicat de intrare; S — instrucțiune (sau secvență de instrucțiuni); R — predicat de ieșire.

Secvențele de apel ale procedurii le vom nota $I_H\{S\}E_H$, unde I_H și E_H sînt predicatele de intrare, respectiv de ieșire. Inițial avem: i:= n numărul de discuri de deplasat; I_0 și E_0 predicatele inițiale și finale.

Din procedura de mai sus se deduc următoarele proprietăți:

1. $I_0 = I_H \wedge (i = n)$
2. $E_0 = E_H \wedge (i = n)$
3. $I_H \wedge (i = 1) = I_x$
4. $E_x = E_t = E_H$
5. $I_H \wedge (i \neq 1) = I_y$
6. $E_z = E_y = I_H$
7. $E_H \wedge (i \neq n) = I_z$ sau I_t

Ultima relație conține o ambiguitate. Aceasta se ridică prin memorarea într-o stivă a adresei apelului recursiv. Pentru a simplifica se introduce o variabilă întreagă *j* prin care se simulează stiva prin înmulțirea și împărțirea ei cu 2 (deplasare de biți la stînga și la dreapta ca într-o stivă). Pe de altă parte se va folosi paritatea acestei variabile pentru a obține adresa apelului

recursiv. Astfel se transformă y, tz și t să cuprindă variabila j în felul următor:

y:: = < j: = j * 2; i: = i — 1; schimbare (b, c) >

z:: = < schimbare (b, c); j: = j + 1; deplasare (a, b); schimbare (a, c) >

t:: = < i: = i + 1; schimbare (a, c); j: = j/2 >

Inițial, j = 1 deci $I_0 \supset (j = 1)$.

Se observă că j este par în E_y și impar în E_z. Se poate deci elimina ambiguitatea din relația 7 astfel:

$$E_H \wedge (i \neq n) \wedge (j \text{ par}) = I_z$$

$$E_H \wedge (i \neq n) \wedge (j \text{ impar}) = I_t.$$

Procedura va avea structura următoare:

sf: = **fals**
j: = 1
atît timp cît ¬ sf **execută** H

unde: $H:: = < \neg sf \wedge I_H\{H\}(I_H \wedge \neg sf) \vee (sf \wedge E_0) >$

Notăm: $E_0 = I_H \wedge sf$, deci $E_H \wedge (i = n) = E_0 = I_H \wedge sf$. Din relațiile 3 și 4 se deduce: $I_H \wedge (i = 1)\{x\}E_H$, iar din relațiile 5 și 6 se obține $I_H \wedge (i > 1)\{y\}I_H$, deci se poate scrie relația

I_H\{**cît timp** i > 1 **execută** y; x\}E_H

Notă:. Ordinul „**execută**" se referă doar la blocul y.

Trei cazuri sînt posibile pentru E_H.

$$E_H \wedge (i = n) = I_H \wedge sf$$

$$E_H \wedge (j \text{ par}) \{Z\}I_H$$

$$E_H \wedge (j \text{ impar}) \wedge (i \neq n)\{t\}E_H$$

de unde se deduce:

E_H\{**cît timp** j impar $\wedge$ (i ≠ n) **execută** t; **dacă** i = n **atunci**
sf = adevărat **altfel** z\}I_H.

Se deduce programul următor:

```
sf: = fals
j: = 1
i: = n
  cît timp ¬ sf execută
                cît timp i > 1 execută y; sfîrșit ciclu
                x;
                cît timp (j impar) ∧ (i ≠ n) execută t; sfîrșit ciclu
                dacă i = n atunci sf: = adevărat altfel z
sfîrșit ciclu
```

S-a obținut un program în care procedura recursivă a fost înlocuită cu alta iterativă.

*

Problema eficienței programelor din punct de vedere timp de execuție și memorie ocupată continuă să rămînă prezentă. Ea nu va fi rezolvată de apariția de calculatoare mai rapide sau cu memorie mai mare. Se observă că odată cu evoluția tehnologică cresc în complexitate și aplicațiile utilizatorului, astfel încît spațiul de memorie necesar este mereu insuficient sau timpul de răspuns este scăzut.

Odată cu apariția minicalculatoarelor problema optimizării programelor a crescut în importanță. Programele mari trebuie rescrise pentru a intra în memoria minicalculatorului.

Transformările actuale care se fac pentru îmbunătățirea performanțelor programului au un caracter empiric și țin de experiența programatorului. Problema optimizării programelor rămîne în continuare în atenția cercetărilor viitoare.

13. Portabilitatea și adaptabilitatea programelor

13.1. Introducere

Un produs-program se spune că este *portabil* dacă poate fi transferat cu ușurință dintr-un sistem de calcul în altul. Evident nu este vorba de sisteme de calcul de același tip ci sisteme diferite care se pot asemăna în unele privințe sau pot fi total diferite.

Problema portabilității programelor devine din ce în ce mai importantă. În condițiile în care se elaborează pachete de programe pentru diverse aplicații economice, industriale, științifice etc. sub un anumit sistem de operare, pentru o anumită mașină de calcul, care evoluează de la un an la altul, s-ar realiza un cîștig important de muncă și energie dacă aceste pachete de programe ar putea fi transferate de la un sistem la altul sau de la o mașină la alta. Se poate spune că după fiabilitate și corectitudine portabilitatea programelor ocupă unul din primele locuri. Un limbaj de programare poate prolifera foarte ușor dacă compilatorul său este portabil.

În țara noastră problema portabilității produselor program începe să se afirme odată cu punerea în exploatare a minicalculatoarelor de tip I-100 sau CORAL. Mulți utilizatori își pun problema dacă pachetele de programe scrise în limbajele COBOL sau FORTRAN pentru calculatoarele FELIX pot fi rulate și sub sistemele de operare ale minicalculatoarelor.

Practic, motivul principal al abordării portabilității programelor, în condițiile unei evoluții tehnologice ascendente, este de a ușura transferarea lor pe un nou calculator sau sub un nou sistem de operare. Dacă efortul de transferare al programatorului este mai mic decît cel cerut pentru realizarea lui pe noul sistem atunci se spune că programul este portabil.

Spre deosebire de portabilitate, *adaptabilitatea* este o măsură a ușurinței cu care un program poate fi modificat pentru a face față noilor cerințe ale utilizatorilor sau ale sistemului. Distincția majoră între cele două concepte este aceea că adaptabilitatea presupune schimbări în structura algoritmului pe cînd portabilitatea este legată de schimbări în mediul programului.

Motivul principal de abordare al adaptabilității decurge din nevoia de a satisface cu un singur program un larg ansamblu de cerințe ale utilizatorului. De exemplu, se dorește extinderea unui program cu noi funcții care nu au fost prevăzute inițial, sau se șterg porțiuni dintr-un program pentru anumiți utilizatori, care nu folosesc anumite facilități.

Deși pare că adaptabilitatea implică portabilitatea totuși există diferențe de principiu. Dacă adaptabilitatea presupune modificări în program, portabilitatea înseamnă elaborarea programului după anumite criterii în așa fel ca la transferarea lui pe o nouă mașină modificările să fie aproape nule.

Dacă se urmărește procesul de realizare al unui program, rezultă că într-o fază oarecare se elaborează și algoritmul de rezolvare a problemei. Algoritmul este independent de felul în care sînt reprezentate datele sau se obțin rezultatele, deși uneori eficiența lui poate să depindă de modul în care se prezintă datele. Un algoritm poate fi realizat pe orice calculator prin implementarea operațiilor sale de bază și a tipurilor de date. Totuși algoritmul original rămîne valabil pentru orice calculator. De aici rezultă un principiu de bază pentru abordarea portabilității, așa cum se va vedea în paragrafele următoare.

În continuare se vor prezenta cîteva tehnici pentru realizarea de produse-program portabile. Cînd un programator dorește să transfere un algoritm pe o nouă mașină trebuie mai întîi să aleagă o reprezentare a operațiilor de bază și a tipurilor de date ale problemei. După aceea trebuie să exprime algoritmul în termenii noii reprezentări. Tehnicile ce vor fi prezentate elimină etapa a doua în întregime.

În ceea ce privește adaptabilitatea programelor nu există tehnici sau procedee bine definite. Pentru realizarea ei, trebuie aplicată o metodă sistematică, de exemplu, trebuie evitată pe cît posibil recodificarea, aceasta fiind o sursă sigură de erori. Se poate aprecia însă gradul de adaptabilitate al unui program. Astfel programele modulare, structurate, în care funcțiile sînt clar delimitate și pot fi regăsite ușor posedă un grad mai mare de adaptabilitate față de programele monolitice.

13.2. Portabilitatea prin limbaje de înalt nivel

Limbajele de înalt nivel nu numai că facilitează procesul de codificare al programului, dar asigură și o anumită portabilitate a lui. Un program scris în FORTRAN sau COBOL poate rula, în principiu, pe orice calculator, sub orice sistem de operare. Deci limbajul de nivel înalt constituie o cale sigură de a asigura portabilitatea produselor program. Trebuie îndeplinite însă unele condiții și anume:

a) Limbajul să aibă o versiune standard și această versiune să fie implementată în toate sistemele de operare. Se poate spune că majoritatea limbajelor de înalt nivel satisfac această condiție (există standarde internaționale (ISO) pentru FORTRAN, COBOL, PL/1, PASCAL etc.) Uneori pentru a exploata mai bine posibilitățile hardware ale mașinii se extinde limbajul prin noi instrucțiuni sau anumite instrucțiuni capătă o nouă formă. Aceste modificări sînt specifice unei implementări locale iar programele care le utilizează se pot executa doar sub sistemul de operare respectiv. De aici rezultă a doua condiție pentru a asigura portabilitatea produselor-program.

b) Trebuie evitate construcțiile specifice unei implementări locale, care nu sînt acceptate de versiunea standard. În acest caz programele se pot transfera practic fără intervenții de la o mașină la alta.

Extensiile unui limbaj pot fi făcute în două feluri:

- fie se definesc noi operații sau tipuri de date cu ajutorul celor care există deja în limbaj; în acest caz programul nu este eficient dar compilatorul rămîne portabil, deci programul este portabil;
- fie se definesc noi operații sau tipuri de date utilizînd facilitățile hardware ale mașinii; în acest caz eficiența programului crește, dar portabi-

litatea sa scade. La transferarea lui pe o nouă mașină trebuie să se țină cont ca aceste extensii să fie realizate folosind noile posibilități hardware. Aceasta implică modificare în cod în compilator, deci sursă de erori.

De exemplu limbajul FORTRAN ANSI nu posedă operații pentru manipularea șirurilor de caractere. Calculatoarele din seria IBM 360 sînt înzestrate cu instrucțiuni speciale (MOVE) pentru prelucrarea șirurilor de caractere. În FORTRAN șirurile de caractere pot fi reprezentate prin tablouri de tip întreg și prelucrate prin operații obișnuite. Această implementare nu este însă avantajoasă pentru calculatoarele IBM 360, pentru că nu se exploatează facilitățile hardware ale mașinii. În schimb, dacă se extinde limbajul FORTRAN prin includerea șirurilor de caractere în tipuri de date și se introduc operații speciale de tip IBM 360 pentru prelucrarea lor, programele ce vor rula pe astfel de calculatoare vor fi mult mai performante, dar neportabile. În scopul de a avea atît programe performante dar și portabile se impune a treia condiție.

c) Operațiile de bază și tipurile de date necesare pentru rezolvarea eficientă a problemei să fie disponibile în limbajul ales pentru codificarea programului. Această condiție este uneori foarte greu de satisfăcut. Rezolvarea multor probleme cere operații și tipuri de date care nu există în limbajul standard, după cum s-a văzut în exemplul de mai sus cu limbajul FORTRAN. De aici rezultă o mare varietate de versiuni ale limbajelor sau chiar o mare varietate de limbaje de programare. Apariția de noi limbaje, ca de exemplu limbajul ADA, este dictată tocmai de necesitatea de a satisface cerințele utilizatorului pentru rezolvarea de noi probleme cu noi tipuri de operații sau structuri de date. Evident că marea varietate de limbaj este o piedică în calea portabilității programelor pentru că, în general, un sistem de calcul nu poate avea în același timp compilatoarele tuturor limbajelor astfel încît să asigure execuția oricărui program, indiferent de limbajul în care a fost scris. Ideal, din punct de vedere al portabilității, ar fi existența unui singur limbaj de nivel înalt standard, care să fie implementat pe toate mașinile de calcul, așa cum limbajul FORTRAN sau COBOL, există aproape în orice centru de calcul. În acest caz programele utilizator ar putea fi transferate de la un sistem la altul fără nici o modificare de cod.

În privința adaptabilității trebuie spus că limbajele de înalt nivel nu rezolvă această problemă. Adaptabilitatea este o problemă de modificare a programului și foarte puține limbaje de nivel înalt posedă mecanisme interne pentru selectarea unor porțiuni de text sursă și ignorarea altora. În general aceste modificări se realizează cu programe speciale, ca de exemplu editoarele de texte. Aceste programe devin din ce în ce mai complexe, tocmai pentru a compensa lipsa mecanismelor de adaptabilitate la nivel de compilator. Astfel editorul de texte EDI din sistemul MIX [8] pentru minicalculatoarele I-100 și CORAL posedă un număr însemnat de comenzi pentru punerea la punct rapidă a programelor sursă. Astfel, de exemplu, prin comanda SAVE se salvează într-un fișier temporar o secvență de instrucțiuni sursă care pot fi inserate apoi prin comanda UNSAVE în orice porțiune a programului. Tot pentru modificarea programelor mai există încă trei programe speciale:

— SLP: program de modificare a textului sursă;

— PAT: program de modificare a codului obiect (rezultatul compilării);

— ZAP: program de modificare a imaginii executabile (rezultatul editării de legături).

Probabil că una din problemele viitoare ale compilatoarelor va fi posibilitatea de a varia codul generat în funcție de anumite opțiuni.

13.3. Portabilitatea prin maşini abstracte

Portabilitatea programului depinde de portabilitatea limbajului, iar portabilitatea acestuia este dependentă de compilator. Dacă un compilator se implementează cu uşurinţă pe orice calculator spunem că limbajul este portabil, deci implicit toate programele scrise în acest limbaj. În acest paragraf vom studia, în particular, problema portabilităţii unui compilator, dar ea rămîne valabilă pentru orice produs software. În mod simplificat rezolvarea unei probleme se poate separa în: o mulţime de operaţii de bază şi tipuri de date; un algoritm pentru manipularea operaţiilor şi a datelor.

Maşina abstractă este un calculator fictiv care posedă specificaţiile de bază şi tipurile de date necesare rezolvării problemei. Ea modelează cerinţele problemei. Algoritmul de rezolvare a problemei se scrie într-un limbaj oarecare pentru acest calculator. Pentru executarea programului pe un calculator real trebuie realizată maşina abstractă pe acest calculator. Se observă că maşina abstractă reflectă tocmai separarea problemei în operaţii de bază, tipuri de date şi algoritm.

Deosebirea între portabilitatea prin limbajul de nivel înalt şi cea prin modelul maşinilor abstracte este evidentă. În primul caz problema se rezolvă într-un limbaj existent, iar portabilitatea programului depinde direct de portabilitatea compilatorului. De aceea la alegerea unui limbaj pentru codificarea unui program se pune mai întîi întrebarea dacă limbajul satisface cerinţele acestuia şi dacă este portabil. Modelul maşinii abstracte presupune construirea unui nou limbaj de nivel înalt ce se traduce apoi pe orice maşină reală, realizîndu-se astfel portabilitatea programului.

Modelul maşinii abstracte este cu atît mai interesat cu cît nu există limbaje adecvate pentru rezolvarea problemelor actuale tot mai complexe. În acest caz utilizatorul îşi creează un model de maşină abstractă specific problemei sale şi apoi îşi concepe un limbaj adecvat pentru programarea acestei maşini. Cu ajutorul unui translator, care este totodată portabil, îşi transpune programul pe o maşină reală pe care îl poate executa. Modelul maşinii abstracte schimbă deci profilul viitorului utilizator, acesta devenind conceptor: el îşi creează limbajul specific pentru rezolvarea problemelor sale.

Se poate spune că alegerea unui limbaj de nivel înalt (FORTRAN, COBOL, etc.) pentru rezolvarea unei probleme înseamnă de fapt selectarea modelului maşinii abstracte a compilatorului, care a fost specificat de conceptorul limbajului, iar extensiile care se fac la limbaj înseamnă modificări ale modelului iniţial pentru a-l face cît mai potrivit pentru o clasă de probleme.

13.3.1. Realizarea modelului de maşină abstractă

Prima etapă în elaborarea unui program prin modelul maşinii abstracte este conceperea modelului. La realizarea lui trebuie avute în vedere următoarele aspecte: (a) relaţia dintre model şi problema de rezolvat; (b) relaţia dintre model şi calculatoarele existente; (c) realizarea translatorului.

Eficienţa programului precum şi portabilitatea lui depind de primele două aspecte. Cu cît modelul este mai apropiat de problemă, sau mai simplu, cu atît este mai puţin eficient dar este portabil: se poate implementa cu uşurinţă pe orice maşină, dar nu se exploatează facilităţile hardware ale maşinii.

Reciproc, dacă modelul reflectă întocmai caracteristicile hardware ale unei maşini, programul va fi foarte eficient pentru maşina respectivă, dar mai puţin portabil. Problema principală este echilibrarea primelor două aspecte. Soluţia la această problemă ar fi conceperea unei ierarhii de maşini abstracte: în vîrful ierarhiei se află o maşină abstractă simplă, care se dezvoltă la nivelul următor, şi aşa mai departe, pînă ce ajunge la nivelul maşinii hardware reale. Această tehnică va fi prezentată în paragraful următor.

La limită poate fi posibil de a construi o maşină abstractă care are o singură instrucţiune, ca de exemplu: calculează cel mai mare divizor comun al numerelor x şi y. Acest model nu este însă interesant, pentru că nu se avansează în nici un fel în rezolvarea problemei. De aceea procesul de realizare a maşinii abstracte începe cu specificarea unor operaţii de bază şi a unor tipuri de date, care apar convenabile pentru rezolvarea problemei. Nu există o regulă pentru selectarea acestora, totul depinzînd de problemă. În general se includ acele operaţii şi tipuri de date cu caracter general, care să permită rezolvarea unei clase de probleme. De obicei se lucrează cu: numere întregi şi operaţii aritmetice; teste de egalitate; operaţii de intrare/ieşire pe caracter.

Se mai pot adăuga, de asemenea: numere reale şi operaţii cu numere reale; şiruri de caractere şi operaţii cu şiruri: concatenare, extragere subşir, comparare etc. operaţii de intrare/ieşire cu zone tampon.

După aceea se defineşte un limbaj simplu ce utilizează aceste tipuri de date. Limbajul cuprinde un set restrîns de instrucţiuni comune tuturor limbajelor de nivel înalt. Structurile de control ale limbajului permit utilizatorului să-şi definească algoritmul de rezolvare a problemei sale. În plus faţă de structurile de control se includ şi instrucţiuni de declarare a datelor, precum şi etichete pentru controlul execuţiei programului. În concluzie, se pot defini: etichete; declaraţii de variabile simple; declaraţii de tablouri; instrucţiuni de salt necondiţionat; instrucţiuni de selecţie; instrucţiuni de realizare a ciclurilor; proceduri şi blocuri de instrucţiuni.

Aşa cum am mai spus, la definirea structurilor de control, a operaţiilor de bază şi a tipurilor de date nu trebuie să se aibă în vedere un singur calculator real. În scopul atingerii portabilităţii conceptorul trebuie să considere caracteristicile unei largi clase de calculatoare. Se poate spune că structura actuală a calculatoarelor este legată în principiu de: organizarea registrelor; organizarea memoriei; mecanismele de adresare a datelor; facilităţile de intrare-ieşire.

În privinţa registrelor există calculatoare fără registre programabile (IBM 1400, IBM 1620), cu un singur registru aritmetic (IBM 7040, IBM 7090, CDC 3000), cu mai multe registre (IBM 360, FELIX-C), sau cu stivă (PDP-11, I-100/102F, CORAL 4011/4030, Burroughs 5000, Burroughs 5500), În legătură cu aceste diferenţe se recomandă ca în modelul maşinii abstracte registrele să nu fie explicit specificate de programator.

În privinţa organizării memoriei, spaţiul de adresare poate fi liniar (IBM 360), sau multiplu (mai multe spaţii de adresare independente: Burroughs 5500), sau memoria poate fi ierarhizată: mai multe nivele de memorie cu diferite viteze de acces (CDC 6000, memorie cache la IBM/85 sau I-102 F).

Adresarea datelor poate fi directă, indirectă sau indexată, iar operaţiile de intrare-ieşire pot fi programate, cu tratare prin întrerupere sau cu acces direct la memorie.

Toate aceste diferenţe hardware fac dificilă realizarea unei maşini abstracte eficientă şi universală. Din punct de vedere al portabilităţii se cere realizarea unei maşini abstracte simple care să poată fi implementată cu

ușurință pe orice calculator. În paragraful 13.3.2 se descrie modelul unei mașini abstracte simple, deci portabilă.

Algoritmul care a fost scris în termenii modelului mașinii abstracte se traduce automat cu ajutorul unui translator. Translatorul trebuie să fie portabil pentru a asigura portabilitatea programelor mașinii abstracte. Evident operația de translatare cuprinde: recunoașterea limbajului sursă; generarea de cod.

Procesul de recunoaștere depinde de limbajul sursă și este diferit de la o mașină abstractă la alta. Cum mașinile abstracte posedă un limbaj foarte simplu și în general se aseamănă între ele elaborarea procedurii de recunoaștere a textului sursă se poate realiza relativ simplu cu un set comun de algoritmi deja consacrați. Problema se pune la generarea de cod pentru că acesta diferă de la o mașină la alta. În momentul în care se transferă translatorul pe o altă mașină trebuie modificat generatorul său pentru a produce cod pentru noua mașină.

13.3.2. Exemplu de mașină abstractă

În acest paragraf vom descrie mașina abstractă INTCODE folosită pentru transferarea compilatorului BCPL pe o nouă mașină. Limbajul BCPL (Bootststrop Combined Programming Language) [44] este un limbaj de nivel înalt pentru programarea de algoritmi caracteristici sistemelor de operare, compilatoarelor, aplicațiilor de calcul științific etc. El posedă un set bogat de structuri de control și permite prelucrări de adrese. Pentru a realiza portabilitatea limbajului s-a construit o mașină abstractă simplă și s-a descris compilatorul în limbajul acestei mașini. Codul obținut constituie interfața dintre partea independentă de mașină a compilatorului și generatorul de cod. Descrierea compilatorului în limbajul mașinii abstracte este foarte simplă, aceasta din motivul că limbajul BCPL fiind conceput pentru codificarea algoritmilor de tip sistem de operare posedă un singur tip de date (întregi), trei tipuri de variabile (globale, dinamice și statice) și un mecanism simplificat de apel a procedurilor.

Mașina INTCODE

Mașina INTCODE [9] este un calculator fictiv cu o structură hardware obișnuită: 6 registre de control, o memorie și o unitate de calcul și control (fig. 13.1).

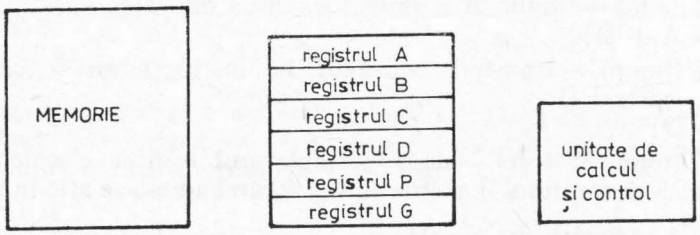


Fig. 13.1. — Componentele mașinii INTCODE.

Memoria este adresată pe cuvinte; lungimea cuvîntului nu este specificată, fiind dependentă de implementarea pe mașina reală. Memoria este văzută ca o succesiune de locații (cuvinte) de dimensiuni egale.

Registrele sînt următoarele:

— A, B: acumulator şi extensia acumulatorului;

— C: contorul de instrucţiuni; conţine adresa următoarei instrucţiuni de executat;

— D: registru de adresă;

— P: pointer pe zona de date locale (registru index);

— G: pointer pe zona de date globale (registru index);

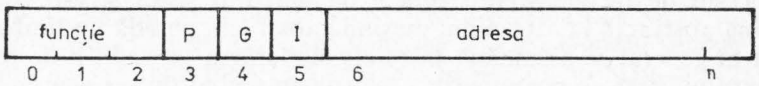


Fig. 13.2. — Formatul instrucţiunii la maşina INTCODE.

Formatul instrucţiunii este următorul (fig. 13.2):

— cod funcţie (3 biţi): conţine unul din cele 8 coduri de instrucţiuni;

— bitul P: specifică dacă conţinutul registrului P trebuie adăugat la registrul D pentru calculul adresei;

— bitul G: specifică dacă conţinutul registrului G trebuie adăugat la registrul D pentru calculul adresei;

— bitul I: indirectare. Dacă este la 1 se încarcă în registrul D conţinutul locaţiei a cărei adresă se află în D;

— cîmpul de adresă (cel puţin 14÷15 biţi); conţine valoarea cu care se încarcă iniţial registrul D.

Instrucţiunea se execută în felul următor: mai întîi se caută instrucţiunea în memorie la locaţia a cărei adresă se găseşte în registrul C, după care C se incrementează. Apoi se încarcă registrul D cu cîmpul de adresă din instrucţiune şi eventual se adună la D conţinutul registrelor P şi G dacă biţii P, respectiv G sînt la 1. Dacă bitul I este la 1 se face indirectarea adresei. În final, cu adresa calculată în D se execută funcţia specifică instrucţiunii. Cele 8 instrucţiuni ale maşinii sînt:

• L (load) — salvează conţinutul acumulatorului în registrul B, după care încarcă A cu adresa din D:

B: = A; A: = D;

• S (store) — stochează acumulatorul la locaţia specificată de D:

(D): = A;

Notă: numele unui registru între paranteze desemnează locaţia a cărei adresă se află în registru.

• A (add) — adună la acumulator adresa din D:

A: = A + D;

• J (jump) — transferă controlul la instrucţiunea a cărei adresă se află în registrul D:

C: = D;

• T (jump if true) — dacă acumulatorul conţine o valoare diferită de zero transferă controlul la instrucţiunea a cărei adresă se află în registrul D:

dacă A ≠ 0 **atunci** C: = D;

• F (jump if false) — dacă acumulatorul este nul transferă controlul la instrucţiunea a cărei adresă se află în D:

dacă A = 0 **atunci** C: = D;

• K (call) — apel de funcție sau procedură. Adresa procedurii se află în registrul A. În D se specifică zona de lucru ce trebuie rezervată în stivă (adresată de P) și se actualizează P să conțină adresa acestei zone. Primele două locații din noua zonă din stivă adresată de P conțin: valoarea anterioară a lui P și vechiul contor de instrucțiuni C. Deci:

D: = P + D
(D) = P
(D + 1): = C
P: = D
C: = A

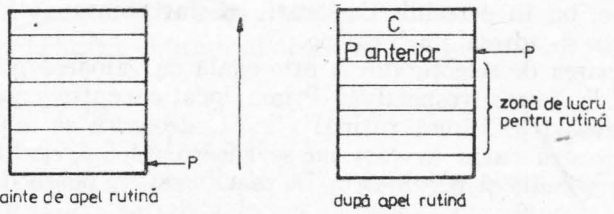


Fig. 13.3. — Starea stivei (înainte și după apel).

În figura 13.3 se prezintă starea stivei înainte și după apel.

• X (execute) — este o instrucțiune ce execută o operație (aritmetică, relațională, logică etc). specificată de conținutul registrului D (codul operației se află în D). Aceste operații sînt următoarele:

X1: A: = (A)
X2: A: = —A
X3: A: = not A
X4: realizează ieșirea (return) dintr-o funcție sau procedură.

Se restaurează vechile valori ale lui P și C:

C: = (P + 1)
P: = (P)

X 5: A: = B * A
X 6: A: = B/A
X 7: A: = B rem A (restul împărțirii lui B cu A)
X 8: A: = B + A
X 9: A: = B — A
X10: A: = B = A

Se testează dacă B și A au același conținut. Dacă da se pune în A valoarea „adevărat" (—1), dacă nu, se pune „fals" (zero). La fel se procedează și pentru următoarele instrucțiuni de acest tip.

X11: A: = B ≠ A
X12: A: = B < A
X13: A: = B ⩾ A
X14: A: = B > A
X15: A: = B ⩽ A
X16: A: = B lshift A

Se deplasează spre stînga configurația binară din B cu un număr de poziții specificat în A;

X17: A: = B rshift A

La fel ca mai sus, dar deplasarea se face spre dreapta;

X18: A: = B $\wedge$ A (și)
X19: A: = B $\vee$ A (sau)
X20: A: = B $\not\equiv$ A (disjuncție)
X21: A: = B $\equiv$ A (conjuncție)
X22: finish — sfîrșit execuție program;
X23: realizează instrucțiunea de tip CASE: în registrul A se află valoarea de selecție, iar în perechile de locații ce sînt pointate de C se află un parametru și adresa unei rutine.
Dacă valoarea de selecție din A este egală cu valoarea parametrului se face salt la rutina respectivă. Prima locație conține numărul de cupluri (parametru, adresă rutină) plus 1, deoarece se mai introduce o pereche pentru cazul în care nu se găsește nici o egalitate între valoarea de selecție și parametru. În pseudocod se poate descrie funcția astfel:

B: = (C)
D: = (C + 1)

pînă ce B = 0 **execută**:
B: = B − 1
C: = C + 2
dacă A = (C) **atunci** [D: = (C + 1); ieșire din ciclu];
sfîrșit ciclu
C: = D

X24, X25 ... se folosesc pentru implementarea dependentă de mașină a operațiilor de intrare-ieșire și a altor funcții speciale.

Limbajul de asamblare INTCODE

Limbajul de asamblare este foarte simplu; vocabularul conține litere, cifre, spații și caracterele „/" și „$".

Caracterul „/" se folosește ca simbol de continuare al unui șir de caractere de la o linie la alta. Scopul este de a avea o utilizare optimă a codului INTCODE pe cartele.

Caracterul „$" se folosește pentru a marca punctele de intrare în rutine.

Instrucțiunile de asamblare ale mașinii INTCODE respectă formatul descris mai înainte și anume:

— o literă (L, S, A etc.) specifică funcția instrucțiunii. Ea este urmată de:

— litera I dacă este necesară indirectarea;
— litera P dacă se adresează o variabilă locală;
— litera G dacă se adresează o variabilă globală;
— adresa, care poate fi un număr zecimal sau o referință la o etichetă. În acest din urmă caz apare litera L (label), urmată de un număr zecimal (eticheta).

Etichetele se specifică prin numere zecimale de la 1 la 500. De exemplu instrucțiunile BCPL următoare:

IF X THEN Y: = 50

V: = V * Y

se traduc în INTCODE astfel:

LIG105 — se încarcă variabila globală nr. 105 (X) în acumulatorul A;
FL10 — dacă A = 0 salt la eticheta 10;
L50 — se încarcă 50 în acumulatorul A;
SP6 — se stochează conținutul acumulatorului (50) în variabila locală nr. 6 (Y);
10 — eticheta 10;
LIP5 — se încarcă variabila locală nr. 5 (V) în A;
LIP6 — se salvează acumulatorul în extensia acumulatorului-B și se încarcă cu variabila locală nr. 6 (Y);
X5 — se face înmulțirea B×A, adică Y×V și se depune rezultatul în A;
SP5 — se stochează rezultatul din A în variabila locală nr 5 (V).

Datele se asamblează cu instrucțiuni de asamblare speciale:

— Dn: se alocă un cuvînt cu valoarea inițială n;
— Cn: se alocă un caracter cu valoarea inițială n.

De exemplu D200 alocă valoarea 200 pe un cuvînt, C20 alocă valoarea 20 pe un octet, iar DL400 alocă pe un cuvînt adresa etichetei 400. Variabilele globale pot fi inițializate în timpul asamblării cu ajutorul directivei „G". În acest caz litera G este urmată de numărul variabilei globale, care la rîndul ei este urmat de litera „L" și un parametru de asamblare. De exemplu G1L2 asignează variabilei globale nr.1 valoarea etichetei nr. 2.

Litera Z se folosește pentru a marca sfîrșitul de cod dintr-un segment. Trebuie spus că existența variabilelor globale permite împărțirea codului INTCODE în mai multe segmente, legătura dintre ele fiind realizată de variabilele globale.

Redăm mai jos exemplul lui Richards [10] de traducere a unei rutine BCPL în INTCODE. Este vorba de o rutină recursivă pentru calculul factorialului unui număr natural. În BCPL rutina apare astfel:

```
GLOBAL $ (START: 1; WRITEF: 76 $)      / definire numere proceduri globale
LET START ( ) BE        $(1            / definirea punctului de intrare în rutină și deschiderea de bloc de instrucțiuni nr. 1
LET F(N) = N = 0 → 1, N * F(N — 1)     /definire funcție recursivă F.
FOR I = 1 TO 10 DO WRITEF(„F(%N) = %N*N", I, F(I))  / imprimarea factorialelor numerelor de la 1 la 10
FINISH )$1                             / terminare execuție și închidere bloc 1
```

Între caracterele „" se specifică formatul de imprimare iar %N este un parametru ce va fi înlocuit cu valoarea următorului argument din lista de argumente. Astfel primul %N se înlocuieşte cu valoarea lui I, iar al doilea %N cu valoarea lui F(I). Traducerea în INTCODE a acestei rutine este următoarea:

```
$1 JL4
$2 LO LIP2 X10 FL6 L1 SP3 JL5 6 LIP2 L1 X9 SP5 LIL3 K3 LIP2
X5 SP3 5 LIP3 X4
4 L1 SP2 JL7 8 LL499 SP5 LIP2 SP6 LIP2 SP9 LIL3 K7
SP7 LIG76 K3 LIP2 A1 SP2 7 LIP2 L10 X15 TL8 X22
3 DL2 499 C11 C70 C40 C37 C78 C41 C32 C61 C32 C37 C78 C10 G1L1
Z
```

Portabilitatea limbajului BCPL

În scopul creşterii portabilităţii compilatorului BCPL se completează acesta cu un generator de cod INTCODE, adică programul BCPL este tradus în instrucţiuni INTCODE. Presupunem că acest compilator BCPL exisă şi în limbaj BCPL şi, de asemenea, că s-a implementat compilatorul pe o maşină oarecare. Dacă se introduce compilatorul BCPL, scris în BCPL, ca text de intrare în compilatorul BCPL se obţine evident compilatorul BCPL în limbaj INTCODE. Pentru transferarea compilatorului BCPL pe o nouă maşină se utilizează: sursa compilatorului în BCPL (cu generare de INTCODE); sursa compilatorului în INTCODE.

Limbajul INTCODE fiind foarte simplu se poate scrie foarte uşor un asamblor şi un interpretor pentru orice maşină.

Richards [8] apreciază la 2 zile realizarea asamblorului şi interpretorului.

Avînd:

— CB: textul sursă în BCPL al compilatorului (ce transformă BCPL în INTCODE);

— CI: textul în INTCODE al compilatorului;

— A, I: asamblorul, respectiv interpretorul de INTCODE, implementate pe maşina pe care se doreşte transferul.

Se cere implementarea compilatorului BCPL pe noua maşină adică să apară compilatorul scris în noul limbaj maşină.

Problema se rezolvă astfel:

a) se asamblează compilatorul CI cu asamblorul A şi se obţine textul INTCODE al compilatorului pe noua maşină — CN;

b) se scrie în BCPL un generator de cod din INTCODE în limbajul noii maşini. Acest lucru este foarte uşor, deoarece limbajul INTCODE este simplu iar elaborarea generatorului se face în BCPL care este un limbaj de înalt nivel. Fie GB acest generator;

c) se transformă generatorul GB în INTCODE cu ajutorul compilatorului CN care se interpretează cu I; se obţine GI;

d) se asamblează GI cu asamblorul A şi se obţine generatorul G în limbaj INTCODE pe noua maşină;

e) se completează compilatorul CN + G care se interpretează cu interpretorul I. Se obţine compilatorul de BCPL, în limbajul noii maşini.

Pe scurt operațiile se pot descrie astfel:

$$CI \xrightarrow{A} CN$$

$$GB \xrightarrow{CN} GI$$

$$GI \xrightarrow{A} G$$

$$CB+GB \xrightarrow{CN+G} C$$

Se observă că limbajul INTCODE rămîne ca limbaj intermediar ceea ce diminuează rapiditatea compilatorului. Dar implementarea lui este foarte simplă: cu excepția asamblorului și interpretorului care sînt simple și chiar se livrează scrise în BCPL odată cu setul compilatorului, nu rămîne de scris decît un generator din INTCODE în limbajul noii mașini.

O altă metodă mai rapidă ca implementare, dar nu rapidă la compilarea programelor, ar fi implementarea compilatorului pe noua mașină în limbaj INTCODE, adică se oprește procesul precedent după faza a).

În acest caz compilarea programelor scrise în BCPL este realizată de către compilatorul CN care se execută prin interpretarea lui de către interpretorul I.

Avantajul utilizării mașinii abstracte INTCODE pentru o nouă implementare a limbajului BCPL este clar: cel care realizează implementarea nu are nevoie să cunoască structura internă a compilatorului BCPL, algoritmii săi de analiză sintexică etc. totul rezolvîndu-se la nivel de limbaj INTCODE.

Așa cum am mai spus simplitatea mașinii INTCODE rezultă și din structura simplă de date a limbajului BCPL; acesta a fost conceput în sensul portabilității. Pentru alte limbaje mai elaborate, ca ALGOL 68 de exemplu, portabilitatea este o problemă mai dificilă, pentru că mașina abstractă corespunzătoare este mai complicată; ea trebuie să reflecte varietatea de tipuri de date și operații de bază ale limbajului.

13.4. Ierarhia de mașini abstracte

Conceptul de ierarhii de mașini abstracte este foarte cunoscut în lucrările de elaborare a sistemelor de programe, ca de exemplu, referitor la protecția informației în astfel de sisteme. În acest paragraf îl vom folosi în sensul producerii de software portabil. Așa cum am mai spus, la elaborarea modelului mașinii abstracte trebuie realizat un compromis între aspectul general al modelului și apropierea lui de un hardware real. Dacă se ține cont de structura unei mașini reale modelul pierde din portabilitate. Acest compromis se poate realiza prin aplicarea conceptului de ierarhii de mașini abstracte.

Ierarhia de mașini abstracte presupune mai multe nivele de mașini abstracte, în care mașinile din vîrful ierarhiei reflectă caracteristicile problemei și prezintă limbaje adecvate pentru codificarea algoritmilor, fiind eficiente dar neportabile. Mașinile de nivel scăzut sînt concepute avînd în vedere structura calculatoarelor; ele sînt portabile, dar neeficiente. Pentru a realiza un echilibru rezonabil între eficiență și portabilitate se alege o mașină abstractă între cele două extreme.

De exemplu, să presupunem că avem de realizat un compilator portabil pentru limbajul X. Prima maşină abstractă pentru rezolvarea acestei probleme poate avea o singură instrucţiune: „compilare X". Pentru implementarea ei pe o maşină reală ar trebui să se scrie întregul compilator în limbajul maşină al calculatorului. Dacă se presupune că dedesuptul acestei maşini se află o altă maşină abstractă, cu mai multe instrucţiuni, procesul de codificare pe o maşină reală exclude de asemenea portabilitatea. În schimb, dacă se continuă

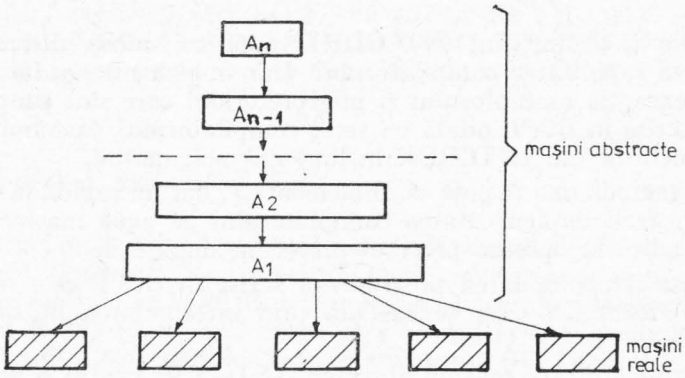


Fig. 13.4. — Model de ierarhie de maşini abstracte.

procesul de rafinare a maşinilor abstracte se poate ajunge la un moment dat la o maşină abstractă cu instrucţiuni de tip INCARCA REGISTRU, STOCHEAZĂ IN MEMORIE, ADUNA, SCADE etc. ce pot fi implementate cu uşurinţă de orice maşină reală. Dacă se codifică compilatorul în limbajul acestei maşini abstracte se poate spune că este portabil. Dar nu se poate spune că este eficient pentru că instrucţiunile în care este codificat sînt universale şi nu ţin cont de facilităţile hardware ale maşinii reale.

Fie ierarhia de maşini abstracte $A_1, A_2, ..., A_n$ (fig. 13.4), structurată astfel încît maşina A_{i+1} este definită în limbajul maşinii A_i, pentru orice i, $1 \leqslant i < n$. Maşina A_n din vîrful ierarhiei asigură limbajul în care se exprimă problema. Maşina A_1 de la baza ierarhiei reprezintă legătura cu mediul exterior, ea poate fi implementată pe o mare varietate de calculatoare reale. Practic se poate spune că maşina A_1 implementează efectiv fiecare maşină abstractă din ierarhie, cu condiţia ca definirea lui $A_i (i = 1)$ în limbajul lui A_{i-1} să nu depindă de caracteristicile maşinilor reale. Astfel se poate spune că orice program scris pentru A_n este portabil. A_n conţine limbajul de definire a problemei, A_1 asigură portabilitatea, iar o maşină abstractă intermediară asigură o implementare portabilă eficientă.

Procesul de implementare a produselor program prin ierarhii de maşini abstracte se aseamănă cu strategia de concepere de sus în jos a programelor (top-down), care constă în expandarea problemei de rezolvat pe un număr oarecare de niveluri pînă se ajunge la elementele logice ce pot fi exprimate direct într-un limbaj de programare. Fiecare nivel din ierarhia de maşini abstracte este echivalent cu un nivel de secvenţe de concepere de sus în jos. Ierarhia de mai sus se poate generaliza astfel ca A_1 să devină bază comună pentru mai multe ierarhii (fig 13.5) .

Aceasta este posibil deoarece A_1 nu are legătură cu caracteristicile problemei de rezolvat. În acest caz deasupra ei se pot afla mai multe ierarhii, cu caracteristici pentru diferite clase de probleme.

Maşina A_1 trebuie să îndeplinească următoarele condiţii:

— să fie simplă astfel încît să poată fi implementată uşor şi repede pe o mare varietate de maşini reale;

— să fie extensibile astfel încît să se poată adăuga noi facilităţi, dacă acestea nu pot fi exprimate uşor cu instrucţiunile existente.

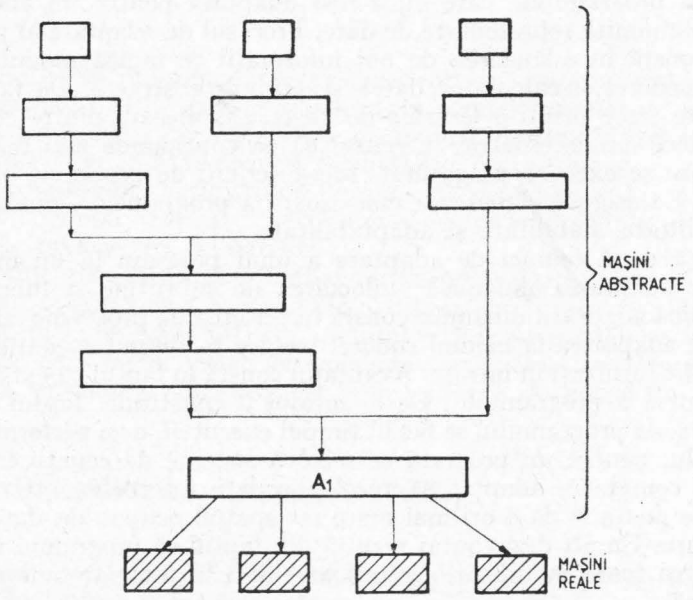


Fig. 13.5. — Model de maşini abstracte cu ierarhie generalizată.

Odată ce A_1 a fost definită, atunci orice program scris într-unul din limbajele maşinilor abstracte ale ierarhiei poate fi uşor transportat de la o maşină reală la alta.

13.5. Adaptabilitatea programelor

Istoric, modificările efectuate în programele elaborate sînt legate mai întîi de schimbările survenite în structura hardware a maşinilor de calcul, în sensul că programele nu exploatau noile facilităţi hardware pentru satisfacerea cerinţelor de performanţă. Uneori apăreau probleme chiar în ceea ce priveşte posibilitatea de execuţie a vechilor programe pe noua maşină.

O altă cauză ce a implicat adaptarea programelor o constituie apariţia şi utilizarea bibliotecilor de rutine partajabile. Punerea la zi a acestor rutine implică uneori modificarea interfeţei cu programul utilizatorului, ceea ce impune intervenţii în acestea din urmă.

Un alt aspect al adaptabilităţii constă în modificarea programului pentru introducerea sau eliminarea unor funcţii realizate de program. În concluzie adaptarea constă în realizarea unor modificări într-un program,

fie pentru a-l putea executa pe o nouă maşină, fie pentru a-l face mai eficient, fie pentru a-i adăuga (suprima) unele funcţii sau pentru a-l integra într-un anumit mediu hardware sau software.

Costul de transformare al programului trebuie să fie minim iar fiabilitatea să rămînă cel puţin aceeaşi. Dificultatea procesului de adaptare rezultă din faptul că se începe modificarea lui fără o bună cunoaştere a programului original.

Efortul de adaptare se reduce foarte mult dacă se porneşte de la o versiune abstractă a programului, care nu a fost adaptată pentru un anumit calculator sau o anumită reprezentare de date. Procesul de adaptare al programului abstract constă în adăugarea de noi informaţii ce indică modul în care se reprezintă concret, în calculator, datele şi acţiunile abstracte. De fapt procesul de adaptare trece printr-o ierarhie de abstracţii, fiecare dintre ele fiind mai concretă decît predecesoarea, în sensul că ea corespunde mai bine mediului final în care se execută programul. Ideea scrierii de programe abstracte nu este nouă. Ea asigură elaborarea mai uşoară a programelor, precum şi o mai bună lizibilitate, fiabilitate şi adaptabilitate.

Există două tehnici de adaptare a unui program la un mediu particular [45]: adaptarea dinamică; înlocuirea de subrutine şi funcţii apelate.

Tehnica adaptării dinamice constă în scrierea de programe cu parametri, astfel încît adaptarea la mediul concret se face în timpul execuţiei în funcţie de parametri furnizaţi în intrare. Avantajul constă în faptul că există o singură versiune sursă a programului. Dezavantajul îl constituie faptul că deciziile de adaptare ale programului se fac în timpul execuţiei, deci performanţa scade. De exemplu, pentru un program ce rezolvă sisteme de ecuaţii în mulţimea numerelor complexe, adaptat să rezolve ecuaţii cu coeficienţi reali timpul de execuţie poate fi de 4 ori mai mare iar spaţiul ocupat de date este de 2 ori mai mare. Un alt dezavantaj rezultă din faptul că programul nu este prevăzut pentru toate modificările ulterioare, deci în final, trebuie modificat şi codul sursă.

Cea de-a doua tehnică, de înlocuire a subrutinelor şi funcţiilor, constă în scrierea acestora în aşa fel încît să manipuleze date abstracte. În final, subrutinele şi funcţiile apelate se modifică în funcţie de cerinţele concrete. Avantajul acestei tehnici constă în faptul că nu cere limbaje sau instrumente speciale de elaborare a programelor, dar presupune modificări în textul sursă al programului abstract. De asemenea, realizarea prelucrării prin apeluri de subrutine şi funcţii face să scadă eficienţa programului. Un exemplu tipic de utilizare a acestei tehnici îl constituie implementarea unei stive. Astfel se definesc funcţii şi subrutine de: iniţializare stivă; testare dacă este vidă; testare dacă este plină; introducere element în stivă; scoatere element din stivă.

Manipularea stivei se face prin apelarea acestor subrutine şi funcţii. În ceea ce priveşte adaptarea unui program ce manipulează stiva, la diferite configuraţii hardware, se pune numai problema reprezentării concrete a stivei. Pentru calculatoarele cu memorie mare stiva poate fi reprezentată printr-un tablou.

Pentru minicalculatoare sau microcalculatoare, cu memorie mai mică, se poate menţine doar o parte a stivei în memorie (cîteva elemente) iar restul se prelungeşte pe un dispozitiv periferic (disc).

Dificultăţile celor două metode pot fi înlăturate prin utilizarea de limbaje extensibile. Aceste limbaje permit declararea de noi tipuri de date şi operatori. Definiţia operatorilor diferă de cea funcţiilor şi subrutinelor prin faptul că selectarea codului de realizare a operaţiei se face în funcţie de tipul operanzilor.

De exemplu, operatorul de înmulțire se poate extinde pentru matrici, a căror înmulțire diferă de cea a scalarilor; algoritmul de realizare a înmulțirii se selectează în funcție de tipul operanzilor; matrici sau scalari. În acest caz adaptarea este de natură statică. Dezavantajul constă în faptul că extensibilitatea, așa cum s-a arătat, este o piedică în calea portabilității programelor. Ca instrumente de lucru pentru adaptarea programelor se pot folosi utilitarele de punere la punct a programelor, ce există în fiecare sisteme de operare: editoare de texte; programul de corecție în imagine obiect; programe de corecție în imagine executabilă.

*

Portabilitatea programelor este dependentă de structura compilatoarelor și de mecanismul utilizat pentru transferarea compilatoarelor de pe o mașină pe alta. În paragrafele precedente s-a prezentat tehnica mașinilor abstracte. Portabilitatea cere însă reducerea facilităților limbajului și evitarea pe cît posibil, în compilator, a funcțiilor dependente de mașină. Limbajul BCPL este portabil tocmai pentru că structura de date este simplă și are puține facilități de manipulare a datelor.

Diversificarea continuă a limbajelor de programare, precum și evoluția în ritm rapid a tehnologiei hardware și a sistemelor de operare ridică tot mai serios problemele portabilității și adaptabilității programelor, de a căror rezolvare va depinde în viitor economisirea unui efort considerabil pentru transferarea produselor program pe noile sisteme de calcul.

13.6. Portabilitatea sistemului UNIX

Sistemul de operare UNIX este un exemplu elocvent de portabilitate a unui produs software realizat într-un limbaj de nivel înalt. Monitorul sistemului de operare și toate utilitarele sistemului—compilatoare, editoare, programe de manipulare fișiere etc.— sînt scrise în limbajul C. În principiu, pentru a instala sistemul UNIX pe un alt calculator este nevoie doar ca noua mașină să posede un compilator pentru limbajul C. În plus trebuie scris un cod foarte redus, în limbajul noii mașini, pentru realizarea operațiilor de intrare-ieșire. Programarea acestui cod cere însă un efort foarte mic. Limbajul C posedă structuri de control asemănătoare limbajelor PASCAL și ALGOL, dar manipulează și entități (adrese, biți) ce permit scrierea de sisteme de operare. Ca și în cazul limbajului BCPL compilatorul de limbaj C generează cod intermediar, ce este apoi tradus în codul de asamblare propriu mașinii.

În plus compilatorul de limbaj C este scris în limbajul C. Deci, realizarea unui compilator C pentru o nouă mașină se reduce la realizarea unui generator de cod mașină din limbaj intermediar C, care implementează, ca și în cazul limbajului BCPL, o mașină virtuală foarte simplă.

Alte compilatoare de limbaje, FORTRAN 77 de exemplu, produc, de asemenea, cod intermediar C. În felul acesta se simplifică foarte mult realizarea și implementarea lor, deoarece nu se coboară pînă la nivelul cel mai de jos al mașinii.

Portabilitatea a asigurat răspîndirea deosebită a sistemului UNIX, devenind astăzi sistem de operare universal, atît pentru minicalculatoare cît și pentru microcalculatoare și calculatoarele mari.

14. Evaluarea și măsurarea performanțelor

14.1. Introducere

Problema evaluării și măsurării performanțelor unui program sau unui sistem de programe este următoarea: cunoscîndu-se specificațiile de definiție ale unui produs-program să se determine dacă modelul de realizare satisface aceste specificații și să se măsoare parametrii caracteristici programului. Deci problema performanței este relativă scopului și obiectivelor produsului respectiv. Nu există un set universal de parametrii pentru a defini performanța, ea referindu-se la parametri și specificațiile sistemului în cauză. Acești parametri fie se deduc (se evaluează) prin analizarea programului (sau modelului) fără a-l executa, fie se măsoară și se înregistrează în timpul execuției programului.

În general performanța se referă la minimizarea sau maximizarea unor parametri ce reprezintă gradul de utilizare a resurselor produsului program. De exemplu: timp de răspuns minim sau capacitate de lucru maximă sînt termeni foarte discutați la studiul performanței unui sistem de operare. Cerințele performanței conținute în obiectivele de concepere ale programului se exprimă prin parametri vizibili utilizatorului (ca de exemplu timpul de răspuns). Conceptorul este interesat însă și de alți parametri interni: lungimea cozilor de așteptare, viteza de încărcare a unui segment etc. Relațiile dintre acești parametri sînt foarte complexe. Problema este de a prelucra parametri interni în așa fel încît parametri externi să corespundă cerințelor utilizatorului.

Se poate vorbi de două niveluri de măsurare a performanței: nivelul minim acceptabil și nivelul optim. Performanța minimă acceptabilă este nivelul la care programul răspunde acceptabil specificațiilor sale. Pentru aceasta, la stabilirea specificațiilor de definire ale sistemului se fixează un set de valori limită pentru anumiți parametri; de exemplu timpul de răspuns la o întrerupere externă să nu fie mai mare de 500 ms. Evident, ideal ar fi ca programul să lucreze la nivelul optim de performanță, adică valorile parametrilor ce exprimă performanța să fie, după caz, minimi sau maximi.

Cînd se măsoară performanța unui sistem trebuie să se facă deosebire între starea lui normală de lucru și stările tranzitorii sau de supraîncărcare. Sistemul va funcționa în cea mai mare parte a timpului în starea sa normală și aceasta este de fapt starea care ne interesează. Totuși trebuie studiate și stările tranzitorii pentru a ști din ce cauză apar și ce influență au asupra sistemului.

Evaluarea și măsurarea performanțelor unui program se face și cu scopul de a îmbunătăți programul. Din studiul unor parametri rezultă unele concluzii ce impun modificări asupra programului în vederea ridicării lui la nivelul optim de performanță.

Trebuie subliniat că îmbunătățirea performanțelor unui program este limitată de factori de natură fizică și economică. În primul rînd sistemele

de calcul utilizează dispozitive fizice, în care viteza de transmitere a semnalului este limitată, deci viteza de execuție a programului este limitată. Pe de altă parte produsele software sînt din ce în ce mai complicate astfel încît niciodată nu funcționează așa cum se dorește: întotdeauna se greșește undeva iar performanța sistemului scade. Complexitatea produselor software poate fi considerată o piedică aproape de netrecut în calea creșterii performanțelor.

O altă limită inerentă rezultă din confilctul dintre obiective, de exemplu cunoscutul conflict spațiu-timp: viteza de execuție poate uneori fi mărită prin consum mai mare de memorie, și invers se poate economisi memorie dar viteza de execuție scade. De exemplu, pentru descompunerea unui număr în factori primi, cel mai rapid ar fi să existe o tabelă în memorie a factorilor primi, dar cum spațiul de memorie necesar ar fi prea mare, practic se utilizează un algoritm care calculează factorii primi, dar timpul de răspuns scade.

Limitele de natură economică sînt legate de costul ridicat al componentelor hardware de înaltă performanță. Tehnologia actuală permite elaborarea de unități de calcul foarte rapide dar și foarte scumpe. De aceea la elaborarea unui proiect se face un compromis între prețul componentei hardware și performanța ei în așa fel încît ambele să fie acceptabile pentru aplicația elaborată. Pe de altă parte și din punct de vedere software, cercetarea celor mai bune soluții presupune timp sau noi resurse, deci un cost ridicat de realizare. De aceea în general se acordă atenție acelor părți din program considerate critice, de care depinde în mare măsură performanța sistemului. Este important ca realizatorul să cunoască în detaliu comportarea întregului sistem și să descopere acele porțiuni critice din sistem pentru care să elaboreze cele mai bune soluții.

14.2. Evaluarea performanțelor

Orice formă de evaluare a performanțelor este aproape imposibilă fără existența unui model. Modelul este o reprezentare abstractă ce conține numai variabilele semnificative și relațiile dintre ele. El poate fi foarte simplu sau foarte complex. Performanța unui sistem nu este un număr, o constantă, ci o funcție sau mai multe funcții ale căror valori depind de parametrii de intrare. În scopul de a caracteriza performanța unui sistem dat trebuie exprimate într-un fel aceste funcții și această expresie constituie modelul sistemului.

Există mai multe tipuri de modele, dar două sînt de bază: modelul logic și modelul analitic. Modelul logic exprimă structura sistemului și poate fi de două feluri: model grafic sau de simulare. Modelul analitic este un set de ecuații matematice ce exprimă relațiile care există între variabilele de bază ale sistemului și parametrii de performanță. Modelul analitic nu reflectă în general structura sistemului, ci numai relațiile dintre variabilele sale.

14.2.1. Modelul grafic

Modelul grafic este foarte des folosit în evaluarea performanțelor unui program. El este un graf orientat ce se deduce din organigrama programului din care se suprimă anumite detalii dar se adaugă noi informații.

Un graf orientat este o mulțime de noduri și arce orientate: fiecare arc pornește de la un nod și se termină la alt nod sau chiar la același nod. Arcele reprezintă acțiuni (instrucțiuni) din program, iar nodurile reprezintă transferul

de control de la o instrucțiune la alta. Informația adițională asociată nodurilor sau arcelor se referă la probabilitatea ca execuția (controlul) să meargă pe arcul respectiv sau să ajungă la arcul respectiv.

Fie secvența de instrucțiuni de mai jos, scrisă în limbajul PL/1:

```
IF  X < 0  THEN X = — X + 1  ELSE  X = X + 1;
IF  Y < 10 THEN Y = Y/2      ELSE  Y = Y/3;
DO  I = 1  TO N;
    TAB(I) = TAB(I) * X — Y;
END;
```

În figura 14.1 se prezintă organigrama programului:

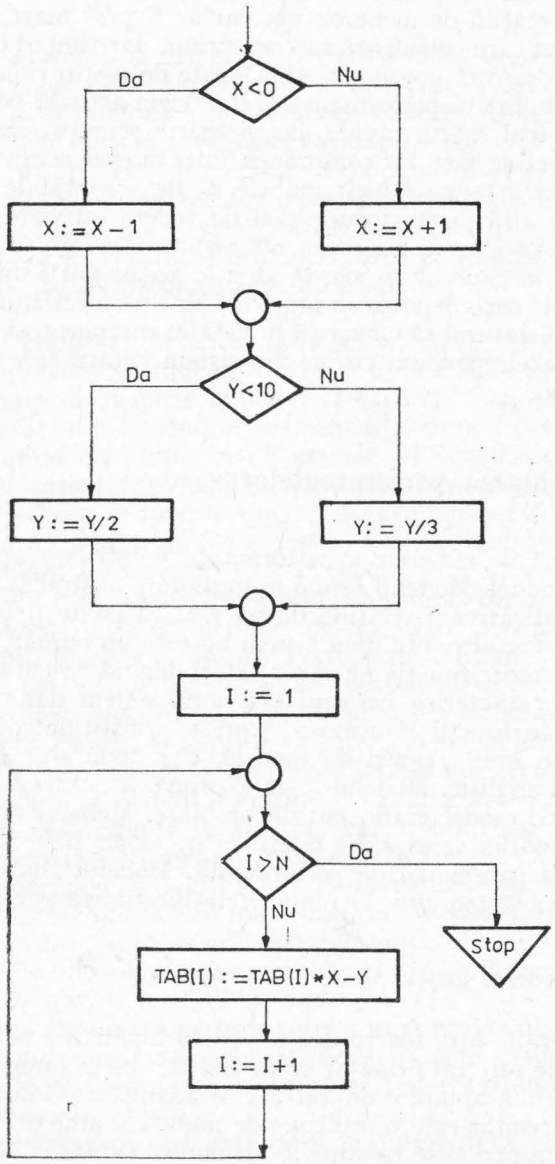


Fig. 14.1. — Organigramă.

Pentru a construi modelul grafic al programului vom face următoarele presupuneri:

— N = 100, X < 0 jumătate din timp, Y < 10 jumătate din timp;

— operațiile de adunare, scădere, comparare, citire din memorie și stocare în memorie durează cîte o unitate de timp fiecare;

— instrucțiunile de înmulțire și împărțire durează 5 unități de timp;

— indexul I se păstrează într-un registru, deci adresarea tabloului nu cere timp suplimentar.

Cu aceste presupuneri se pot face următoarele estimări:

Operații	*timp de execuție*
X < 5	3: citire operanzi și comparare
X = —X + 1	5: citire operanzi, schimbare semn, adunare și stocare
X = X + 1	4: citire operanzi, adunare, stocare
Y < 10	3: citire operanzi și comparare
Y = Y/2	8: citire operanzi, împărțire, stocare
Y = Y/3	8: citire operanzi, împărțire, stocare
I = 1	1: încărcare registru
I > N	3: citire operanzi și comparare
TAB(I) = TAB(I) * X — Y	10: citire operanzi, înmulțire, scădere, stocare
I = I + 1	1: incrementare registru

În figura 14.2 se reprezintă graful orientat al programului în care pe fiecare arc se scrie probabilitatea de execuție a arcului și timpul său de execuție.

Acest model este foarte simplu și trebuie extins pentru a putea reprezenta programe mai complicate. O primă situație pentru care modelul este inadecvat este cazul în care N este variabil. Dacă N nu este cunoscut se poate estima valoarea lui medie. Dacă totuși variația lui N este mare în raport cu valoarea lui medie se poate folosi același model dar probabilitatea determinată de N devine parametru, de exemplu $p = N/(N + 1)$. Apare, de asemenea, problema apelului de subrutine sau proceduri, pentru că timpul de execuție al apelului depinde de numărul de parametrii de intrare. Se poate propune un model în care apelul de subrutină să fie reprezentat printr-un arc pe care timpul de execuție se reprezintă prin valoarea sa medie și varianța sa [2].

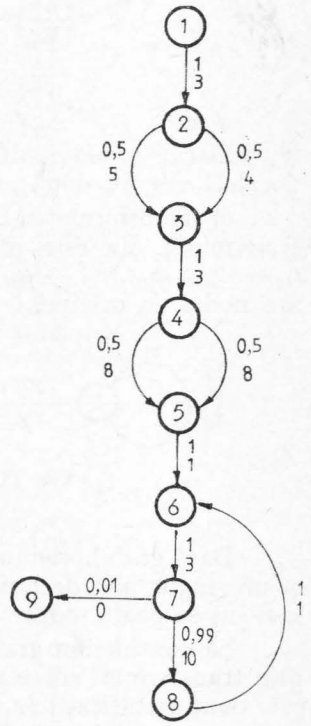


Fig. 14.2. — Graf orientat.

14.2.2. Evaluarea performanței cu ajutorul modelului grafic

Pentru evaluarea performanței programului cu ajutorul modelului grafic se face mai întîi o reducere a complexității grafului prin aplicarea mai multor transformări elementare. Graful obținut va fi echivalent cu cel original numai din punct de vedere al execuției programului și nu în ceea ce privește structura lui. Se definesc 3 transformări elementare [2].

a) Transformarea serie se aplică unor perechi de arce în secvență, adică nodul terminal al unui arc este nod origine pentru celălalt arc. Perechea de arce precum și nodul dintre ele se înlocuiesc cu un singur arc, ca în figura 14.3.

Fig. 14.3. — Model de transformare serie.

Se notează cu p_{xy} probabilitatea ca execuția să se facă pe arcul (xy) și cu t_{xy} timpul de execuție pe arcul (x, y). În cazul transformării serie (fig. 14.3), avem: $p_{ij} = p_{ik}p_{kj}$ și $t_{ij} = t_{ik} + t_{kj}$.

b) Transformarea paralelă se aplică unei perechi de două arce în paralel, adică cele două arce au același nod origine și același nod terminal. Transformarea paralelă este ilustrată în figura 14.4, unde $p_{ij} = p'_{ij} + p''_{ij}$ și $t_{ij} = (p'_{ij}t'_{ij} + p''_{ij}t''_{ij})/(p'_{ij} + p''_{ij})$.

Fig. 14.4. — Model de transformare paralelă.

Dacă sînt mai mult de două arce paralele se aplică succesiv transformarea paralelă pentru două arce pînă se ajunge la un singur arc.

c) Transformarea ciclului se aplică unui arc ce are același nod și origine și terminal. Ea este prezentată în figura 14.5, unde $p_{ij} = p'_{ij}/(1 - p_{ii})$ și $t_{ij} = t'_{ij} + p_{ii}t_{ii}/(1 - p_{ii})$. Aceste valori se calculează pentru fiecare arc ce are nodul ca origine.

Fig. 14.5. — Model de transformare a ciclului.

Dacă graful orientat al programului nu are decît un singur nod de intrare și un singur nod de ieșire, atunci prin transformările elementare prezentate mai sus se poate reduce la un singur arc.

Să considerăm graful programului precedent (fig. 14.6) și să-l reducem prin transformări succesive. Se aplică mai întîi transformarea paralelă pentru cele două ramificații ce pornesc din nodul 2, respectiv 4, obținîndu-se graful prezentat în figura 14.7. Se calculează noua probabilitate și noul timp de execuție pentru arcele rezultate:

$$p_{23} = 0{,}5 + 0{,}5 = 1, \qquad t_{23} = (0{,}5 \cdot 5 + 0{,}5 \cdot 4)/(0{,}5 + 0{,}5) = 4{,}5$$

$$p_{45} = 0{,}5 + 0{,}5 = 1, \qquad t_{45} = (0{,}5 \cdot 8 + 0{,}5 \cdot 8)/(0{,}5 + 0{,}5) = 8.$$

Se aplică transformarea serie pentru nodurile de la 1 la 6 și se calculează (fig. 14.8) $p_{16} = 1$, $t_{16} = 3 + 4{,}5 + 3 + 8 + 1 = 19{,}5$.

Se aplică transformarea serie pentru secvențele de noduri: 7, 8, 6 și se calculează (fig. 14.9):

$$p_{76} = 0{,}99 \cdot 1 = 0{,}99 \qquad t_{76} = 10 + 1 = 11.$$

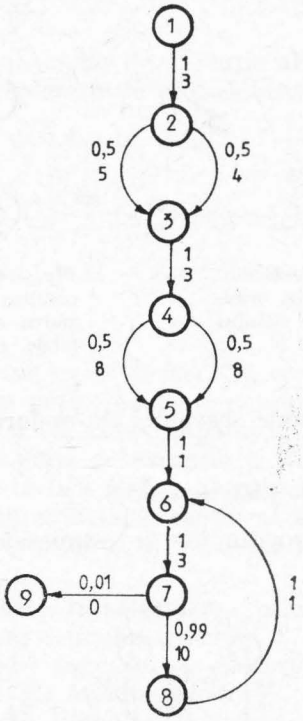


Fig. 14.6 — Graful orientat al unui program.

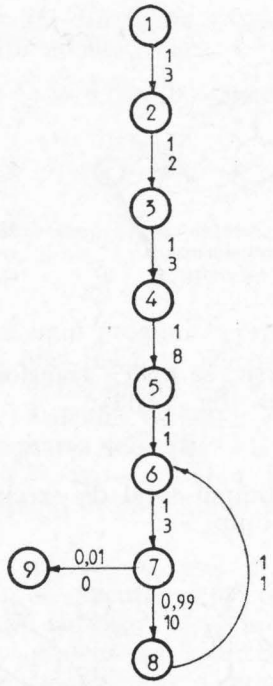


Fig. 14.7. — Graful rezultat prin transformarea paralelă

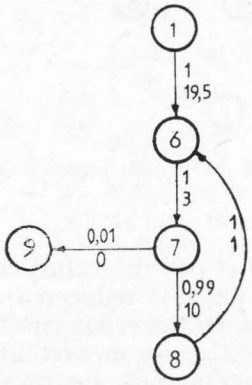


Fig. 14.8. — Graful rezultat prin transformarea serie.

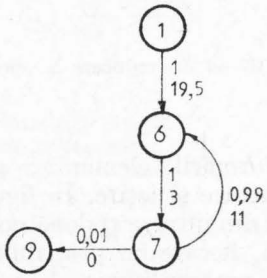


Fig. 14.9 .— Graful rezultat prin transformarea serie pentru secvența de noduri 7, 8, 6.

Se aplică transformarea serie pentru secvența de noduri 6, 7, 6 și se calculează (fig. 14.10):

$$p_{66} = 1 \cdot 0{,}99 = 0{,}99, \qquad t_{66} = 3 + 11 = 14.$$

Se aplică transformarea ciclului nodului 6 și se calculează (fig. 14.11):

$$p_{69} = 0{,}01/(1 - 0{,}99) = 1, \quad t_{69} = 0 + 0{,}99 \cdot 14/(1 - 0{,}99) = 1386$$

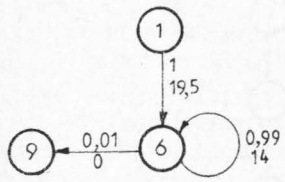


Fig. 14.10. — Graful rezultat prin transformarea serie pentru secvența 6, 7, 6.

Fig. 14.11. — Graful rezultat prin transformarea ciclului nodului 6.

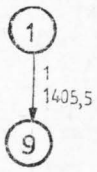


Fig. 14.12. — Graful rezultat prin transformarea serie a secvenței de noduri 1, 6, 9

În sfîrșit, se aplică transformarea serie secvenței de noduri 1, 6, 9 și se calculează (fig. 14.12):

$$p_{19} = 1, \qquad t_{19} = 19{,}5 + 1386 = 1405{,}5.$$

Deci timpul total de execuție al programului se estimează la 1 405,5 unități de timp.

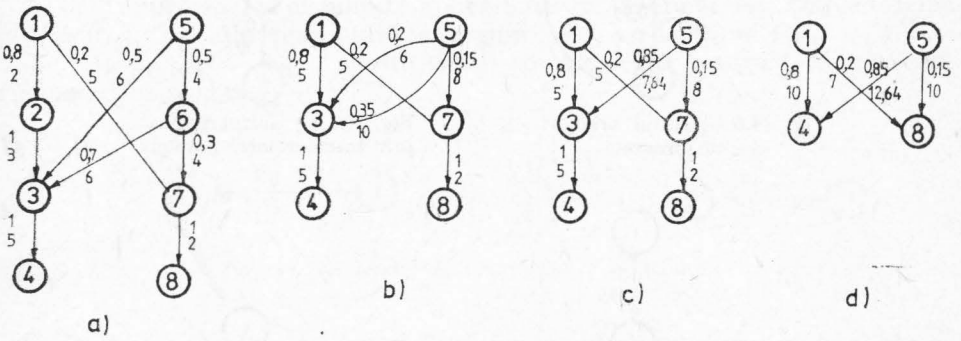


Fig. 14.13. — Model de reducere a unui graf cu două noduri de intrare și două noduri de ieșire

Transformările elementare se pot aplica și pentru grafuri cu mai multe noduri de intrare și ieșire. În figura 14.13 se prezintă reducerea unui graf cu două noduri din intrare și două noduri de ieșire. Graful redus din figura 14.13 *d* are 4 arcuri, fiecare cu probabilitatea lui de a atinge un nod de ieșire. Dacă se cunoaște probabilitatea de a intra în program prin fiecare din nodurile sale de intrare se poate calcula timpul mediu de execuție al programului. De exemplu, să presupunem că probabilitatea de a intra în nodul 1 este 0,6 iar

pentru nodul 5 este 0,4. Probabilitatea de a atinge nodul de ieşire se poate calcula pentru fiecare arc:

arc	*probabilitate*	*timp de execuţie*
(1,4)	$0{,}6 \cdot 0{,}8 = 0{,}48$	10
(1,8)	$0{,}6 \cdot 0{,}2 = 0{,}12$	7
(5,4)	$0{,}4 \cdot 0{,}85 = 0{,}34$	12,64
(5,8)	$0{,}4 \cdot 0{,}15 = 0{,}06$	10

Timpul mediu de execuţie al programului din figura 14.13 *d* este dat de suma ponderată de probabilităţi a tuturor arcelor, adică:

$$t = 0{,}48 \cdot 10 + 0{,}12 \cdot 7 + 0{,}34 \cdot 12{,}64 + 0{,}06 \cdot 10 = 10{,}537.$$

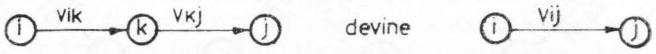


Fig. 14.14. — Model de calcul a varianţei timpului de execuţie pentru transformarea serie.

Aşa cum arată Beizer [2], comportarea unui program reprezentat prin modelul său grafic poate fi exprimată mai bine dacă se calculează pentru fiecare arc şi varianţa timpului de execuţie. Astfel graful redus va indica şi varianţa timpului de execuţie pe fiecare arc. Calculul varianţei v a timpului de execuţie se face astfel: a) pentru transformarea serie (fig. 14.14): v_{ik} este varianţa timpului de execuţie al arcului (i, k), v_{kj} — varianţa timpului de execuţie al arcului (k, j), v_{ij} — varianţa timpului de execuţie al arcului (i, j), $v_{ij} = v_{ik} + v_{kj}$;

b) pentru transformarea paralelă (fig. 14.15): p'_{ij} este probabilitatea de parcurgere a arcului superior, t'_{ij} — timpul de execuţie al arcului superior, v'_{ij} — varianţa timpului de execuţie al arcului superior, p''_{ij} — probabilitatea de parcurgere a arcului inferior, t''_{ij} — timpul de execuţie al arcului inferior, v''_{ij} — varianţa timpului de execuţie al arcului inferior, t_{ij} — timpul de execuţie al arcului (i, j) după reducere, v_{ij} — varianţa timpului de execuţie al arcului (i, j) după reducere, $v_{ij} = (p'_{ij}v'_{ij} + p''_{ij}v''_{ij} + p'_{ij}t'^2_{ij} + p''_{ij}t''^2_{ij})/(p'_{ij} + p''_{ij}) - t^2_{ij}$;

Fig. 14.15. — Model de calcul a varianţei timpului de execuţie pentru transformarea paralelă.

c) pentru transformarea ciclului (fig. 14.16): p_{ii} este probabilitatea de parcurgere a ciclului, t_{ii} — timpul de execuţie al ciclului, v_{ii} — varianţa timpului de execuţie al ciclului, v'_{ij} — varianţa timpului de execuţie al arcului (i, j), v_{ij} — varianţa timpului de execuţie al arcului (i, j) după reducere, $v_{ij} = v'_{ij} + v_{ii}p_{ii}/(1 - p_{ii}) + t^2_{ii}p_{ii}/(1 - p_{ii})^2$.

Reluăm graful din figura 14.13 pe care vom reprezenta și varianța timpului de execuție. În plus pentru a reduce graful la un singur arc vom introduce un nod fictiv de intrare de la care pornesc arce spre nodurile de intrare cu anumite probabilități (de exemplu, 0,6 pentru nodul 1 și 0,4 pentru nodul 5) și un nod fictiv de ieșire (fig. 14.17). Prima cifră de pe fiecare arc reprezintă

Fig. 14.16. — Model de calcul a varianței timpului de execuție pentru transformarea ciclului.

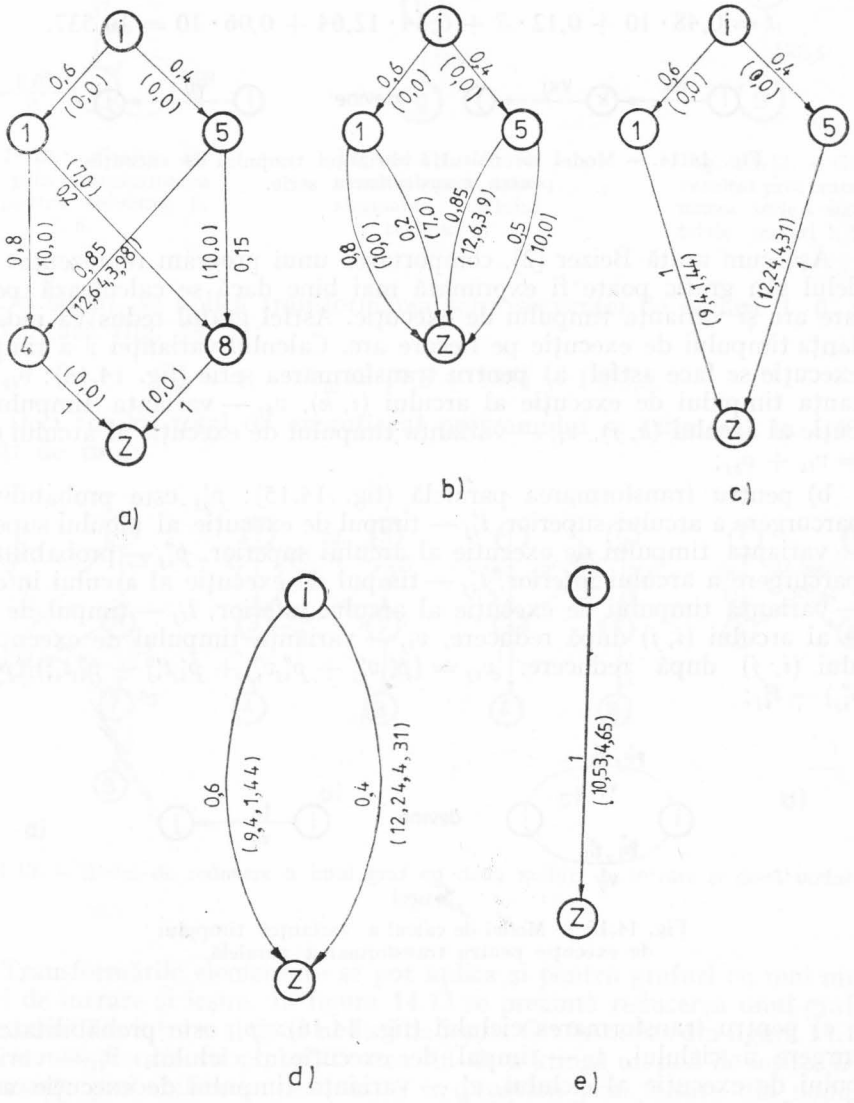


Fig. 14.17. — Reprezentarea varianței timpului de execuție a grafului din figura 14.13.

probabilitatea de parcurgere a arcului, iar între paranteze se află timpul mediu de execuție al arcului urmat de varianța sa. Inițial varianța este nulă. Singura transformare care produce varianța nenulă este cea paralelă din figura 14.13*b*

$$v_{53} = (0{,}5 \cdot 0 + 0{,}35 \cdot 0 + 0{,}5 \cdot 36 + 0{,}35 \cdot 100)/(0{,}5 + \\ + 0{,}35) - 7{,}64^2 = 3{,}98.$$

În final în figura 14.17 *e* se prezintă timpul mediu de execuție al programului și varianța sa.

Modelul grafic prezentat mai sus este util pentru estimarea timpului mediu de execuție fie al întregului program, fie al anumitor porțiuni din program. Totuși această estimare nu este posibilă decît pe baza unor presupuneri, ca de exemplu numărul de execuții ale unui ciclu. De asemenea, nu putem preciza întotdeauna probabilitatea de parcurgere a fiecărui arc. O rezolvare a acestor probleme ar fi simularea. În acest caz se extinde modelul grafic pentru a cuprinde și distribuția probabilităților de execuție a arcelor și, după toate reducerile posibile, se folosește pentru simulare [3].

14.2.3. Evaluarea structurilor de control

Evaluarea timpului de execuție al programului se poate face pornind direct de la structurile sale de control fără a mai construi modelul grafic.

Fiind dat un algoritm *a* notăm cu t_a timpul său de execuție. Fie structurile de control corespunzătoare programării structurate:

a) Secvența *a*; *b*. Dacă t_a și t_b sînt timpii de execuție pentru *a*, respectiv *b*, atunci timpul *t* de execuție al secvenței *a*; *b* este: $t = t_a + t_b$.

b) Selecția: dacă *c* atunci *a* altfel *b*. Notăm cu *p* probabilitatea ca predicatul *c* să fie adevărat. Dacă t_a, t_b, t_c sînt timpii de execuție pentru *a*, *b*, respectiv *c*, atunci timpul de execuție al selecției este: $t = t_c + pt_a + (1 - p)t_b$.

c) Iterația: atît timp cît *c* execută *a*. Timpul de execuție al ciclului este:

$$t = t_c + \frac{p}{1-p}(t_a + t_c).$$

Dacă $p = n(n+1)$, atunci $t = t_c + n(t_a + t_c)$, unde *n* este numărul de iterații.

Fie, de exemplu, programul următor analizat în paragraful precedent:

```
IF X < 0  THEN X = - X + 1 ELSE X = X + 1;
IF Y < 10 THEN Y = Y/2     ELSE Y = Y/3;
DO I = 1 TO N;
TAB (I) = TAB(I) * X - Y;
END
```

Dacă se calculează timpul de execuție al fiecărei structuri de control avînd în vedere estimările din paragraful precedent se obțin următoarele rezultate:

— prima instrucțiune IF: $3 + 0{,}5 \cdot 5 + 0{,}5 \cdot 4 = 7{,}5$
— a 2-a instrucțiune IF: $3 + 0{,}5 \cdot 8 + 0{,}5 \cdot 8 = 11$
— ciclul DO: $3 + 100\ (10 + 3) = 1\ 303$.

În total execuția programului se estimează la 1 321,5 unități, ceea ce reprezintă o abatere în minus de 5,9% față de rezultatul obținut la evaluarea modelului grafic. Aceasta rezultă din faptul că iterația este reprezentată într-un mod mai apropiat de realitate în cazul modelului grafic.

14.2.4. Simularea

Simularea este în totalitate o tehnică experimentală. Gordon [4] o definește ca modul de a rezolva o problemă urmărind schimbările în timp ale unui model dinamic asociat problemei. În esență se observă modul în care variabilele modelului evoluează cu timpul, după care se deduc relațiile dintre aceste variabile.

Sînt două tipuri importante de simulare: continuă și discretă. Modelul unui sistem continuu este în general un set de ecuații diferențiale. Calculatoarele analogice sînt cele mai adecvate pentru acest tip de simulare.

În cazul simulării discrete sîntem interesați de valorile variabilelor numai în anumite perioade de timp și anume după apariția unor „evenimente". Schimbările în sistem sînt provocate doar de aceste evenimente. Sistemele de operare, pachetele de programe pentru diverse aplicații etc. sînt prin esență sisteme discrete. În continuare ne vom limita la tipul de simulare discretă. Un eveniment este un moment în timp în care apare o schimbare în starea sistemului. În general evenimentele sînt provocate de activitățile ce se execută în cadrul sistemului: terminarea unui program, terminarea unei operații de intrare-ieșire etc. Timpul dintre evenimente, de obicei, se ignoră.

Pentru simularea discretă trebuie să existe: un sistem de simulare; un program utilizator pentru simularea unor funcții specifice modelului.

În general sistemul de simulare cuprinde: un limbaj special de modelare; un translator sau interpretor pentru limbaj; un set de rutine.

Utilizatorul își descrie modelul în limbajul de modelare. Această descriere fie se interpretează direct, fie se traduce într-un program care realizează simularea atunci cînd se execută. Sistemul de simulare asigură utilizatorului mijloace de a colecta date, de a le analiza, precum și facilități de listare astfel încît poate observa cu ușurință schimbările variabilelor din model în scopul de a deduce relațiile dintre ele.

Trebuie subliniat că timpul constituie conceptul fundamental pentru simulare. Timpul de simulare nu are de obicei legătură cu timpul real; el poate fi actualizat la apariția evenimentelor. Pentru aceasta se menține un ceas de simulare care avansează la timpul la care va apare următorul eveniment. Deci ceasul nu este la curent de timpul dintre evenimente, ceea ce este normal pentru că nimic nu intervine între evenimente. Timpul de simulare constituie baza relațiilor de timp din model. O funcție majoră a sistemului de simulare este de a determina timpul la care trebuie să apară fiecare eveniment. Metoda cea mai simplă de a realiza aceasta este de a menține o listă a evenimentelor ordonată după timpul lor de apariție. Viitorul eveniment

va fi primul eveniment din listă. O altă metodă este de a calcula timpul de apariție al următorului eveniment pe baza distribuției probabilității lor de realizare.

De exemplu, să considerăm cazul unui sistem de operare în care evenimentele sînt constituite de sosirea lucrărilor (job-uri) ce trebuie executate. Calea cea mai simplă de a descrie modul de sosire al lucrărilor este intervalul de timp între două sosiri succesive. Dacă acest interval de timp nu variază el va fi o constantă; dacă variază, ceea ce este cazul în practică, va fi descris de o distribuție de probabilități. Este practic de a defini distribuția de sosire $A(t)$ a lucrărilor ca probabilitatea pe un interval să fie $>t$. Cum, $F(t)$, funcția cumulativă a distribuției este probabilitatea ca intervalul $\leqslant t$ rezultă: $A(t) = 1 - F(t)$.

În practică sosirea lucrărilor este complet aleatoare. Dacă se impune condiția ca media ratei de sosire (frecvența) să fie o constantă λ, atunci distribuția intervalelor de timp este o exponențială

$$A(t) = e^{-\lambda t},\ t \geqslant 0.$$

Funcția de densitate a intervalului de timp este

$$f(x) = \lambda e^{-\lambda t},\ t \geqslant 0.$$

Probabilitatea ca n sosiri să aibă loc în intervalul t este

$$p(n) = (\lambda t)^n e^{-\lambda t}/n!\ \text{(distribuție Poisson)}$$

Funcția cumulativă a distribuției devine

$$F(x) = 1 - A(t) = 1 - e^{-\lambda t},$$

iar inversa ei

$$\lambda t = -\ln(1 - F(x)).$$

Sosirea unei lucrări constituie un eveniment. Cînd ceasul de simulare atinge valoarea de producere a evenimentului o nouă lucrare se introduce în sistem. Pentru a calcula timpul de sosire a lucrării următoare se procedează astfel: se generează un număr aleatoriu în intervalul (0, 1), iar cu funcția inversă $\lambda(t)$ a funcției $F(t)$ se calculează intervalul de timp după care trebuie să apară noua lucrare. Timpul de sosire a noii lucrări va fi egal cu timpul curent plus intervalul calculat. Este relativ simplu de a genera secvențe de numere aleatoare uniform distribuite în intervalul (0, 1) și multe sisteme posedă rutine pentru acest lucru.

Modelul Poisson prezentat mai sus este larg utilizat în practică. Dacă coeficientul de variație σ/μ (σ este deviația standard, iar μ este valoarea medie) ce măsoară gradul în care datele sînt dispersate în jurul mediei este mult mai mic ca 1 sau mai mare ca 1 trebuie utilizate distribuții Erlang sau hiperexponențiale.

La descrierea modelului și a altor informații necesare pentru simulare se folosesc instrumente de programare adecvate, cum ar fi limbajele de simulare. Ele permit descrierea de clase de entități, atribute, activități, evenimente precum și distribuții de probabilități pentru colectarea de date. Culegerea de date și analizele efectuate asupra lor depind de model și de scopul simulării. De exemplu, pentru analiza unui algoritm de alocare a memoriei interne se măsoară: memoria ocupată m_i de fiecare lucrare; valoarea medie; deviația standard; variația lungimii cozii de așteptare la memorie: valoare medie, deviație standard, maxim, minim; timpul de așteptare în coadă etc.

Sînt două clase de limbaje de simulare:

— *generale* ce permit simularea unui mare număr de sisteme dinamice: sisteme de operare, sisteme economice, aplicații de timp real etc.;

— *speciale* ce se folosesc pentru un anumit tip de sistem, ca de exemplu sistemul de operare.

Cele mai cunoscute limbaje de simulare generale sînt: GPSS, SIMSCRIPT, SIMULA, CSL [5], SIMUB.

14.2.5. Modelul analitic

Modelul analitic este un set de ecuații matematice ce exprimă relațiile care există între variabilele de bază ale sistemului și parametrii de performanță.

Numărul modelelor analitice este destul de mare, dar toate posedă o proprietate comună, și anume conțin variabile stohastice. Acestea se definesc în mod obișnuit prin funcții care specifică probabilitatea cu care iau valori într-un interval dat.

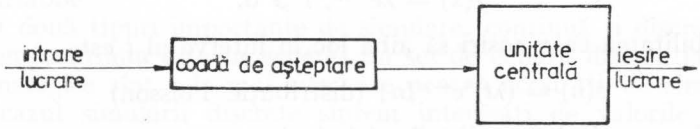


Fig. 14.18. — Modelul FIFO în varianta cea mai simplă.

Ca exemplu de model analitic simplu se poate da modelul „primul sosit primul servit" (FIFO: first-in-first-out) folosit în sistemele de operare pentru gestiunea lucrărilor. În varianta cea mai simplă modelul conține un singur procesor (unitate centrală) și o singură coadă de lucrări (fig. 14.18).

Unitatea centrală prelucrează întotdeauna lucrarea din vîrful cozii (prima sosită) și o execută pînă se termină. Lucrarea terminată părăsește sistemul și procesorului i se alocă o nouă lucrare din vîrful listei de așteptare. Dacă lista este vidă, unitatea centrală trece în așteptare pînă ce sosește o nouă lucrare. Această strategie de alocare a unității centrale se folosește foarte des în sistemele de prelucrare în loturi fără multiprograme. Pentru a construi modelul analitic trebuie specificat timpul la care fiecare lucrare intră în sistem și timpul de execuție al lucrării. Metoda uzuală este de a specifica distribuția probabilității de sosire a lucrărilor și de execuție a lor. De exemplu se poate presupune o distribuție de sosire după legea lui Bernoulli sau binomială. Se poate presupune că timpul de execuție al unei lucrări este un multiplu al unei cuante de timp Q și este dat de distribuția geometrică

$$p_n = (1 - \sigma)\sigma^{n-1}, \quad n = 1, 2, 3, \ldots, \quad 0 \leqslant \sigma < 1,$$

unde p_n este probabilitatea ca timpul de execuție al lucrării să fie exact n cuante, adică nQ.

Un model mai complicat este bazat pe strategia de execuție planificată în timp (round-robin), utilizată în sistemele de timp partajat. Și în acest model o lucrare care sosește se pune la sfîrșitul cozii și procesorului i se alocă lucrarea din vîrful cozii. Spre deosebire de metoda precedentă, lucrarea nu se execută decît o cuantă de timp Q, după care se pune la sfîrșitul cozii în așteptare de

o nouă cuantă de timp, și așa mai departe, pînă se termină și părăsește sistemul (fig. 14.19).

Aceleași distribuții de probabilități pot fi folosite ca și pentru modelul precedent.

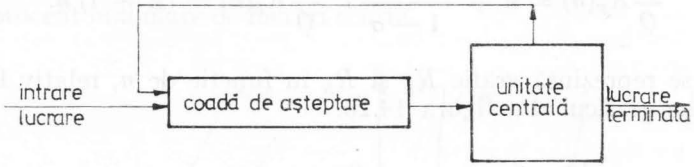


Fig. 14.19. — Model FIFO în varianta strategiei de lucru planificată.

Există și alte modele analitice: alocarea unității centrale pe bază de priorități, paginare, alocare pe disc etc. [6]. În continuare ne vom rezuma la o prezentare comparativă a modelelor FIFO și round-robin.

Evaluarea unui model analitic

Pentru exemplele precedente de modelare FIFO și round-robin vom presupune că lucrările sosesc conform cu distribuția discretă a lui Bernoulli cu probabilitatea λQ, unde Q este cuanta de timp, iar λ este numărul mediu de lucrări ce sosesc pe secundă. De asemenea, presupunem că timpul de execuție al unei lucrări corespunde distribuției geometrice:

$$p_n = (1 - \sigma)\sigma^{n-1}, \quad n = 1, 2, 3, \ldots, \quad 0 \leqslant \sigma < 1,$$

unde p_n este probabilitatea ca timpul de execuție să fie exact de n cuante (nQ secunde). În ambele modele numărul mediu de lucrări este dat de formula

$$E = \rho\sigma/(1 - \rho),$$

unde $\rho = \lambda Q/(1 - \sigma)$, λ este numărul mediu de lucrări ce sosesc pe secundă, $1/(1 - \sigma)$ este numărul mediu de cuante cerute pentru execuția unei lucrări, Q este numărul de secunde pe cuantă, ρ — timpul mediu de execuție, cerut pe secundă, de toate lucrările din sistem; E tinde la infinit, iar ρ tinde la 1.

Pentru modelul FIFO timpul de răspuns este

$$R_F(n) = QE/(1 - \sigma) + nQ,$$

$R_F(n)$ exprimă timpul total pe care o lucrare îl cheltuiește în sistem știind că cere n cuante de timp pentru execuție. Timpul de execuție este nQ, la care se adaugă $QE/(1 - \sigma)$ secunde pierdute în coada de așteptare. Pentru modelul round-robin timpul de răspuns este [1]

$$R_R(n) = \frac{nQ}{1 - \rho} - \frac{\lambda Q^2}{1 - \rho}\left[1 + \frac{(1 - \sigma(\sigma + \lambda Q))(1 - (\sigma + \lambda Q)^{n-1})}{(1 - \sigma)^2(1 - \rho)}\right].$$

O bună aproximare este

$$R_R(n) \simeq nQE + nQ,$$

unde nQ este numărul de cuante necesare pentru execuție, iar nQE este numărul de secunde pierdute în coada de așteptare. Se observă că relațiile de mai sus sînt liniare în n, deoarece Q, σ, λ sînt constante. Ele se mai pot scrie și astfel:

$$\frac{1}{Q}R_F(n) = n + \frac{E}{1-\sigma}, \quad \frac{1}{Q}R_R(n) = (E+1)\,n.$$

Dacă se reprezintă grafic R_F și R_R în funcție de n, relativ la factorul $1/Q$ se obține graficul din figura 14.20.

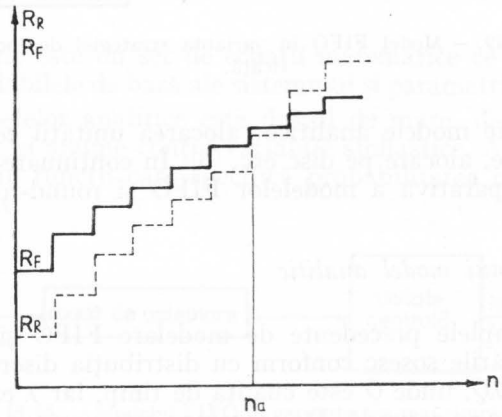


Fig. 14.20. — Reprezentarea grafică a variabilelor R_R și R_F.

Valoarea lui n_a în care se intersectează cele două grafice se calculează astfel

$$n_a + E/(1-\sigma) = (E+1)n_a, \text{ de unde } n_a = 1/(1-\sigma).$$

Începînd de la n_a timpul de răspuns al strategiei FIFO devine mai bun (scurt) decît al strategiei round-robin. Aceasta înseamnă că metoda round-robin asigură un timp de răspuns mai bun pentru lucrările scurte ($n < n_a$). De exemplu, fie $\sigma = 0,1$, deci $p_1 = (1-\sigma) = 0,9$, adică probabilitatea ca timpul de execuție al unei lucrări să fie de o cuantă este de 0,9. În acest caz $n_a = 1,1$. Cum 90% din lucrări au timpul de execuție de o cuantă este bine să se folosească strategia round-robin care asigură un timp de răspuns mai bun.

Se mai poate studia comportarea acestor modele dacă sistemul se apropie de condițiile de supraîncărcare, adică ρ se apropie de 1. Ne vom interesa la timpul pierdut în coada de așteptare. El este egal cu timpul de răspuns din care se scade timpul de execuție:

$$\alpha C_F(n) = \alpha(R_F(n) - nQ), \quad \alpha C_R(n) = \alpha(R_R(n) - nQ),$$

unde $\alpha = (1-\sigma)/(\sigma Q)$ este o valoare de normalizare pentru ca timpul de așteptare în coadă să fie o funcție numai de ρ. În figura 14.21 se prezintă curbele $\alpha C_F(n)$ și $\alpha C_R(n)$ pentru două valori ale lui σ și mai multe valori pentru n [1].

Din figura 14.21 rezultă că timpul de răspuns se deteriorează dacă ρ crește: cu cît crește timpul de utilizare a unității centrale cu atît crește și timpul de așteptare în coadă. Pentru strategia de round-robin serviciul se deteriorează mult mai repede dacă se execută lucrări cu timp mare de execuție. Această deteriorare este mai severă pentru valori mici ale lui σ, adică pentru un procent mai mare de lucrări scurte.

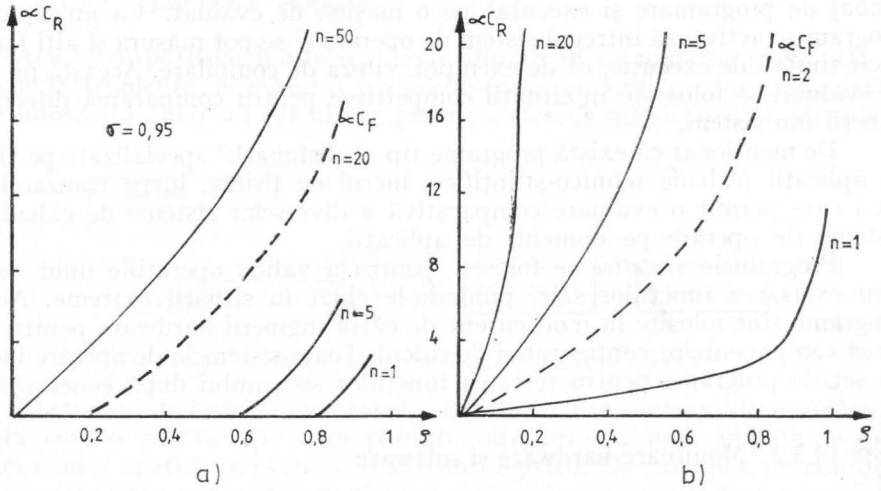


Fig. 14.21.— Reprezentarea grafieă a curbelor $C_F(n)$ și $C_R(n)$.

14.3. Măsurarea performanțelor

Măsurarea performanțelor unui program sau unui sistem de programe se poate face din mai multe motive: se verifică dacă programul satisface specificațiile sale; se dorește îmbunătățirea programului; se dorește compararea programului cu alte variante.

Pentru aceasta se folosesc mai multe tehnici din care reținem două: se rulează un set de programe caracteristice; se execută măsurători prin procedee hardware sau software.

14.3.1. Kernel, benchmark, program sintetic

Prin kernel se desemnează un program special ce se execută pentru a măsura timpul său de execuție. El poate trata diverse probleme ca: actualizare de fișiere secvențiale, cu acces direct etc.; sortare de fișiere; inversare de matrice; evaluare de polinoame etc.

Rezultatele se colectează în tabele după care se listează. Astfel printr-un set de 36 de lucrări distincte rulate sub același sistem de operare s-au comparat calculatoarele IBM 360/65 și IBM 370/165, urmărindu-se determinarea raportului vitezei de calcul. Măsurîndu-se timpul de utilizare a unității centrale pentru fiecare program și făcîndu-se media s-a ajuns la un raport de 3,67 în favoarea calculatorului 370/165.

Pentru a realiza o bună comparare a două sisteme este necesar ca parametrii problemei să fie bine specificați și configurațiile de calcul să posede aceleași caracteristici (dimensiune, memorie, număr canale etc.). Pe de altă parte structura fișierelor și anumite detalii de codificare se pot lăsa nespecificate pentru a da posibilitatea programatorului să folosească toate avantajele sistemului sau mașinii.

Prin benchmark se desemnează un program codificat într-un anumit limbaj de programare și executat pe o mașină de evaluat. Cu un astfel de program se activează întregul sistem de operare și se pot măsura și alți factori decît timpul de execuție, ca de exemplu viteza de compilare. Această metodă de evaluare se folosește în situații competitive pentru compararea diferitelor funcții din sistem.

De menționat că există programe tip „benchmark" specializate pe tipuri de aplicații (calcule tehnico-științifice, lucrul cu fișiere, lucru tranzacțional etc.) care permit o evaluare comparativă a diverselor sisteme de calcul sau sisteme de operare pe domenii de aplicații.

Programele *sintetice* se folosesc pentru a valida operațiile unui sistem prin exersarea funcțiilor sale, punîndu-le chiar în situații extreme. Aceste programe sînt folosite în mod curent de către inginerii hardware pentru testarea componentelor configurației de calcul. Toate sistemele de operare includ un set de programe pentru testarea funcțiilor sistemului după generarea sa.

14.3.2. Monitoare hardware și software

Mărimile ce se pot determina, referitor la un sistem de calcul, se pot clasifica pe trei niveluri:

— *nivel utilizator*: timpul de execuție al programelor, timpul de execuție al unor faze (compilare, editare de legături), memorie utilizată, număr de cartele citite, număr de linii imprimate etc.;

— *nivel sistem*: alocare de resurse, activitatea canalelor, cozi de așteptare etc.;

— *nivel instalație*: fluxul lucrărilor, intervențiile operatorului, cerințele utilizatorului etc.

O parte din aceste date se observă direct, altele se deduc din modele analitice sau de similare. Problema care se pune este de a stabili lista cantităților de măsurat, de a decide asupra instrumentelor de măsurare, de a stabili cît de frecvent se culeg datele, cum se listează datele și mai ales ce analiză trebuie făcută asupra lor.

Monitoarele hardware și software sînt instrumente cu care se observă, se înregistrează sau se măsoară date sau evenimente din sistemul de programare. Monitorul hardware este un dispozitiv fizic, electronic, ce se cuplează la sistemul de calcul. Monitorul software este un program sau o rutină ce se include în pachetul de programe. Fie că este hardware sau software monitorul are următoarea structură: componenta de tratare a întreruperii pentru culegere date (sonda); componenta de înregistrare și analiză a datelor (analizatorul); componente de control care sincronizează activitatea monitorului cu a sistemului de observat; componenta de listare (afișare) a rezultatelor.

Fiecare clasă de monitoare are avantajele și dezavantajele ei. Astfel monitoarele hardware (fiind dispozitive fizice) sînt mai ușor de introdus în sistem. De asemenea, ele nu afectează încărcarea sistemului (nu produc overhead) și pot fi utilizate continuu sau selectiv. În schimb sînt limitate în posibilitățile de măsurare.

Monitoarele software sînt mai flexibile și pot fi folosite pentru observarea unor funcții interne din sistem care nu sînt accesibile monitoarelor hardware (ca de exemplu gestiunea cozilor de așteptare la o partiție). În schimb ele cer alocare de resurse (fișiere, tabele pentru înregistrare date) și interferează cu sistemul, încetinind activitatea acestuia.

14.3.2.1. *Monitoare hardware*

Aceste dispozitive hardware devin din ce în ce mai complexe. De la primele instrumente de măsurat, ca osciloscoape și contoare s-a ajuns astăzi să se folosească chiar un calculator pentru a efectua măsurători pe alt calculator.

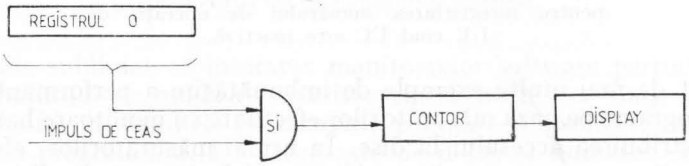


Fig. 14.22. — Modelul unui contor într-un monitor hardware.

Elementul de bază al monitorului hardware este contorul de eveniment. Acesta este o poartă „și" care permite înregistrarea unui impuls de ceas atunci cînd a apărut un eveniment într-un registru. De exemplu, cu contorul din figura 14.22 se poate afișa de cîte ori registrul va atinge valoarea maximă.

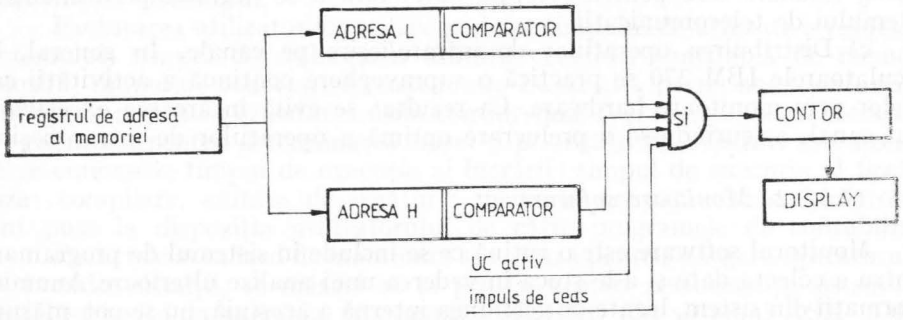


Fig. 14.23. — Schemă de bază a unui monitor hardware pentru înregistrarea numărului de execuție al unei instrucțiuni dintr-o zonă dată de memorie.

În figura 14.23 se prezintă schema de bază a unui monitor hardware ce înregistrează numărul de ori de execuții al unor instrucțiuni dintr-o zonă dată de memorie: $M = \{x \mid L \leqslant x \leqslant H\}$

În figura 14.24 se prezintă schema logică a unui monitor hardware pentru înregistrarea numărului de operații de I/E atunci cînd unitatea centrală este inactivă.

Tot cu monitoare hardware se poate recunoaște execuția unei instrucțiuni și astfel se pot face statistici asupra frecvenței de utilizare a instrucțiunilor calculatorului. Cu monitorul din figura 14.23 se poate măsura execuția instrucțiunilor din zona rezidentă a sistemului de operare, obținîndu-se informații asupra încărcării sistemului (overhead).

Un exemplu de monitor hardware complex este contorul de măsurare a activității sistem, componentă a calculatorului IBM 370/165, care permite selectarea pentru măsurare a uneia din următoarele 7 funcții: operații de

intrare/ieşire; paralelismul operaţiilor de intrare/ieşire; calcul şi operaţii de intrare/ieşire; neoperaţional; lucrul în mod supervizor; lucrul în mod utilizator; timp de calcul total.

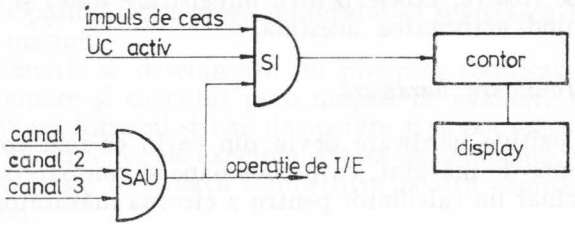


Fig. 14.24. — Schema unui monitor hardware pentru înregistrarea numărului de operaţii de I/E cînd UC este inactivă.

Se pot da mai multe exemple de îmbunătăţire a performanţelor sistemelor de programe pe baza măsurătorilor efectuate cu monitoare hardware [7]:

a) Distribuirea accesului la disc. În urma măsurătorilor efectuate la 5 unităţi de disc s-a constatat că o unitate era folosită excesiv. Prin transferarea unor biblioteci de programe şi pe alte unităţi de disc s-a redus timpul de acces al unităţii în cauză, îmbunătăţindu-se performanţa sistemului.

b) Analiza unui sistem de telecomunicaţie. S-a măsurat timpul de execuţia a instrucţiunilor din porţiunea de memorie destinată prelucrării mesajelor şi prin comparare cu numărul de mesaje sosite s-a redus frecvenţa de polling (scanare linii pentru recepţie date) fără a se degrada performanţele sistemului de telecomunicaţii.

c) Distribuirea operaţiilor de intrare/ieşire pe canale. În general, la calculatoarele IBM 370 se practică o supraveghere continuă a activităţii canalelor prin monitoare hardware. Ca rezultat se evită încărcarea excesivă a unui canal, asigurîndu-se o prelucrare optimă a operaţiilor de intrare/ieşire.

14.3.2.2. *Monitoare software*

Monitorul software este o rutină ce se include în sistemul de programare pentru a colecta date şi a le stoca în vederea unei analize ulterioare. Anumite informaţii din sistem, legate de gestiunea internă a acestuia, nu se pot măsura decît cu monitoare software. De exemplu, numărul de lucrări ce se execută în aceeaşi partiţie sau lungimea unei cozi de aşteptare sînt informaţii care nu sînt accesibile monitoarelor hardware.

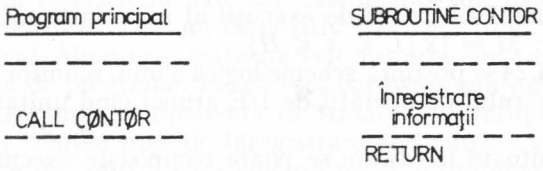


Fig. 14.25. — Model de integrare rutină pentru analiza unui algoritm.

Pentru aceasta se construieşte o rutină ce se introduce în algoritmul a cărui comportare dorim s-o analizăm. În timpul execuţiei algoritmului se apelează rutina de control, care înregistrează anumite informaţii prezente în momentul apelului şi redă controlul apelantului (fig. 14.25).

De cele mai multe ori monitorul software se reduce la un simplu contor ce se incrementează ori de cîte ori se produce evenimentul așteptat (fig. 14.26).

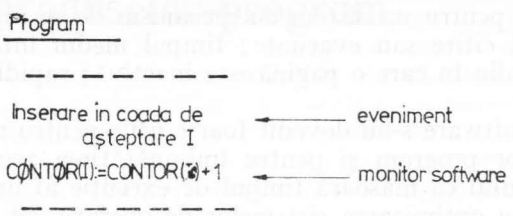


Fig. 14.26. — Schema monitorului software redus la un contor pentru înregistrarea numărului de apariții ale evenimentului așteptat.

Trebuie subliniat că inserarea monitoarelor software perturbă comportarea programului de analizat: monitoarele software cer timp și spațiu de execuție. Astfel în sistemele cu memorie paginată se poate schimba traficul de pagini al programului. De aceea este de dorit ca numărul de monitoare precum și informațiile pe care le înregistrează, să fie cît mai reduse posibil.

Actual noțiunea de monitor a fost extinsă; se cunosc trei tipuri de monitoare software: (a) rutina de contabilizare a lucrărilor, care colectează date referitoare la operațiile executate de programele utilizatorului; (b) pachete standard de monitoare, care se rulează periodic pentru a obține informații în caz de situații speciale; (c) rutine scrise în conjuncție cu conceperea și dezvoltarea sistemului.

Facturarea utilizatorilor se face pe baza timpului de utilizare a resurselor sistemului: timpul de utilizare a unității centrale, cantitatea de memorie folosită, timpul de conectare la sistem etc. Pe de altă parte, aceste informații sînt necesare pentru orientarea utilizatorilor cum să-și reducă costul lucrărilor. De exemplu, pentru determinarea clasei și priorității unei lucrări este necesar să se cunoască: timpul de execuție al lucrării; timpul de execuție al fiecărei faze: compilare, editare de legături; memoria ocupată. Aceste informații sînt puse la dispoziția utilizatorului de către programele de contabilizat.

Pachetele standard de monitoare se rulează pentru a sugera operatorului, în anumite situații, ce măsuri să ia pentru a mări performanța sistemului. Astfel se poate da exemplu programul RMDEMO din sistemul RSX-11M al firmei DEC, care afișează la un display starea curentă a sistemului: memoria ocupată, numele programelor active, fragmentarea memoriei, fragmentarea zonei de alocare dinamică.

Un alt exemplu este sistemul de măsuri software al lui Boole și Babbage [7], care cuprinde următoarele programe:

— PPE (Probleme Program Efficiency): se află în aceeași partiție cu programul utilizator și se activează la fiecare 1/60 s pentru a înregistra timpul de unitate centrală folosit la execuția de instrucțiuni din afara unei zone date de memorie;

— DSO (Data Set Optimizer): înregistrează mișcările capului de disc și sugerează reorganizarea datelor în așa fel încît să se reducă timpul mediu de acces;

— SUPERMON: elaborat pentru sistemul OS/360 MVT. În plus față de cele spuse mai sus dă posibilitatea de a observa utilizarea memoriei: memorie ocupată, memorie liberă, fragmentarea memoriei. Monitoarele speciale se elaborează odată cu fazele de concepere ale sistemului și se folosesc pentru

măsurarea și analizarea unor funcții interne specifice sistemului, ca timpul de execuție al unor porțiuni „critice", eficiența algoritmilor de alocare a resurselor periferice sau memoriei etc.

De exemplu, pentru un sistem cu mecanism de paginație se măsoară: numărul de pagini citite sau evacuate; timpul mediu între două erori de pagină; timpul mediu în care o pagină este inactivă; rapiditatea memoriilor asociative etc.

Monitoarele software s-au dovedit foarte utile pentru măsurarea performanțelor produselor program și pentru îmbunătățirea acestora. În primul rînd, datorită faptului că măsoară timpul de execuție al unui program, sînt necesare atît pentru optimizarea sistemelor de operare cît și a programelor utilizator.

Monitoarele speciale sînt foarte utile pentru generarea secvențelor de test ale noului sistem. Contoarele plasate în diferite porțiuni ale sistemului indică gradul de utilizare al diverselor rutine din sistem. Dacă există porțiuni din program unde contorul este nul sau foarte mic rezultă că secvența de test nu este completă. Monitoarele software asigură de asemenea informații suficiente pentru detectarea erorilor. Un contor plasat într-un ciclu înregistrează numărul de iterații și se poate verifica dacă acesta este corect.

Totuși interesul pentru monitoarele software rezultă din aportul lor deosebit pentru evaluarea performanțelor produselor program. Prin măsurarea timpului de execuție al fiecărei porțiuni din program se poate ști precis dacă programul se încadrează în specificațiile stabilite. De asemenea, studiind frecvența de execuție a instrucțiunilor se poate optimiza acea porțiune de cod care este mai des executată. Unele statistici arată că doar 3% din instrucțiuni ocupă 50% din timpul de execuție al programului. De asemenea, se estimează că monitoarele pot fi concepute în așa fel încît încărcarea sistemului să fie de numai $1 \div 5\%$, ceea ce este destul de mică, deci se poate permite utilizarea lor pe perioade relativ lungi.

15. Conducerea activității de elaborare a produselor-program

15.1. Introducere

Programarea constituie o activitate complexă care presupune folosirea unor importante resurse materiale, de forță de muncă și bănești și care duce în final la obținerea unor produse-program.

Ca și în celelalte domenii de activitate este necesară etapizarea procesului de elaborare a produselor-program și organizarea procesului de „fabricație" după metode care să permită obținerea unor produse-program de calitate superioară și cu cost redus.

Se cere, de asemenea, o metodologie științifică de planificare, urmărire și control a modului de utilizare a resurselor în procesul elaborării produselor-program.

După părerea noastră neglijarea aspectelor de planificare, programare, coordonare, comandă, urmărire și control în munca de programare poate avea implicații negative în planul costurilor și al eficienței.

Problema etapelor ce trebuie parcurse în procesul elaborării am prezentat-o pe larg în capitolul 1. S-a reținut ca principale etape: definirea cerințelor sistemului; definirea cerințelor software; proiectarea de ansamblu; proiectarea de detaliu; codificarea, depanarea și testarea componentelor; integrarea și testarea produselor-program; implementarea (punerea în lucru); exploatare și întreținere; cu precizarea că fiecare etapă (fază) se termină printr-o *activitate de validare și verificare* avînd ca obiectiv eliminarea anomaliilor din produs chiar din momentul în care au fost introduse. Produsele realizate într-o etapă servesc ca punct de plecare pentru dezvoltările ulterioare, ceea ce înseamnă preocuparea pentru a asigura încă de la început calitate și exactitate.

Structura pe etape devine recursivă cînd elaboratorul efectuează modificări cu ocazia elaborării unor noi versiuni.

A rezultat că dezideratul ingineriei programării este ca în procesul elaborării unui produs-program etapele să se succeadă pe cît posibil secvențial fără reveniri semnificative în fazele anterioare.

Se știe însă că în practică acest lucru este de cele mai multe ori imposibil.

Faptul că nu există încă un cadru metodologic care să permită formalizarea definirii cerințelor program și analiza automată a coerenței și corectitudinii acestora face ca în practică să nu se poată realiza parcurgerea etapelor decît cu reveniri, care produc uneori modificări importante.

Organizarea conducerii activității de elaborare a produselor-program mari are la bază desfășurarea acestora conform modelului în cascadă, în fiecare fază activitățile sînt definite folosind conceptul de componentă de lucru, componenta caracterizîndu-se printr-un produs care este apoi subiectul unei evaluări tehnice.

La organizarea întregii activități de elaborare a produselor-program trebuie însă avut în vedere faptul că faza de *fabricație* coincide cu cea de *cercetare-dezvoltare*, lipsind faza de *fabricație de serie*. În schimb, distribuirea și întreținerea programelor pun probleme mai complicate decît a celorlalte produse industriale. Produsul-program este unul singur, dar multiplicat *) în „oglindă" ajunge la sute sau mii de utilizatori. În aceste condiții este primordială păstrarea integrității lui deoarece modificările, care de altfel se pot face relativ ușor asupra acestor „imagini", pot anihila orice efort de întreținere și dezvoltare, întrucît fabricantul de produse lucrează cu originalul, despre care presupune că este identic cu toate copiile aflate în exploatare.

15.2. Activități specifice programării

Dacă analizăm fazele ciclului de viață al programelor desprindem ca principale activități specifice programării: întocmirea specificațiilor și structurarea problemei pe module; coordonarea activității de elaborare a programelor (procedurilor) corespunzătoare fiecărui modul; elaborarea programelor pe module; testarea și integrarea programelor într-un tot unitar care constituie produsul-program; efectuarea controlului tehnic de calitate (pe etape ale procesului de elaborare a programelor); gestionarea colecției de programe, a versiunilor acestora și rularea programelor pe calculator; elaborarea documentației care împreună cu programele definesc împreună produsul-program; dezvoltarea și adoptarea programelor (dacă aceeași unitate asigură și dezvoltarea sau adoptarea programelor la cerințele mai multor beneficiari).

Facem de la început precizarea că în stadiul actual aceste activități nu constituie și specialități de sine stătătoare. Așa cum vom vedea în paragrafele următoare un programator realizează mai multe activități. În funcție de natura sarcinilor realizate vom obține și o structură a personalului care are în sarcină elaborarea unui produs-program.

Nefiind vorba de o activitate de producție nu se poate face nici o delimitare precisă pe categorii. Se înțelege că activitățile de coordonare, control, testare și integrare nu pot fi realizate decît de un programator cu înaltă calificare și vastă experiență în programarea calculatoarelor.

Pentru estimarea timpului și a necesarului de forță de muncă se iau în considerare următorii factori: obiectivele programului; natura activității; mărimea programului; limbajul de programare; baza de date; standardele existente în programare.

Statisticile în dezvoltarea de produse-program evidențiază o analogie strînsă celei de dezvoltare a echipamentelor.

De exemplu, se consideră că este necesar un efort de două ori mai mare pentru a efectua o instrucțiune program decît pentru a efectua o poartă logică hardware (AND, OR, NOT etc.).

În baza unui studiu comparativ al instrucțiunilor unui program complex pentru lucru în timp real (estimate la 160 000) și al punților logice al sistemului de echipamente (estimate la 170 000) s-a constatat că costurile de întreținere a proiectului per instrucțiune le depășesc pe cele de întreținere a proiectului per punte logică printr-un factor de 4: 1.

---

*) Operația de multiplicare se realizează cu cheltuieli minime întrucît ea constă în reproducerea produselor-program în mai multe exemplare prin simplă copiere.

Același studiu estimează că sistemele de programe au complexitate egală cu cea a sistemelor de echipamente.

Costurile mai ridicate ale produselor-program sînt cauzate de faptul că pe plan metodologic procedurile folosite în realizarea echipamentelor sînt mult mai avansate.

Proiectanții hardware au în plus o mult mai mare experiență și folosesc un desen mult mai bine structurat.

Totodată blocurile de construcții de bază folosite în proiectarea produselor program au fost mai numeroase și mai complexe decît blocurile simple de construcții folosite în hardware.

Folosirea programării structurate apropie considerabil cele două domenii în sensul că ambele au la bază un set minim de structuri de bază.

Iată și motivul pentru care considerăm deosebit de important folosirea unei metode științifice care să permită, pe activitățile sus menționate, obținerea unor componente de calitate superioară.

15.3. Sistemul de conducere

Din cuprinsul lucrării a rezultat că activitatea de programare este desfășurată de către un colectiv de programatori și că ea are multe din caracteristicile activităților de cercetare-dezvoltare.

Pe aceste considerente s-a optat pentru sistemul conducerii pe bază de proiecte, care are o durată de acțiune limitată la ciclul de viață al unui produs-program.

În literatura de specialitate [71, 72] se indică multiple modalități de aplicare practică a sistemului conducerii pe bază de proiecte în funcție de amploarea proiectului; noutatea proiectului; durata perioadei de realizare; numărul și potențialul personalului unității, personalitatea conducătorului de proiect etc.

În [71] se indică două modalități de conducere pe bază de proiecte, pe care le considerăm importante și pentru activitatea de elaborare a produselor program dintre care evidențiem:

A. *Conducerea pe bază de proiect cu responsabilitate individuală* caracterizată prin atribuirea întregii responsabilități pentru elaborarea produsului-program unei singure persoane. Pentru realizarea unor subactivități șeful de proiect solicită persoane aflate în compartimente diferite. Această modalitate poate fi avantajoasă numai pentru produse-program simple care nu justifică constituirea unei echipe de elaborare. Chiar și în aceste condiții apar unele probleme, mai ales privind calitatea lucrărilor efectuate și încadrarea în termenele prestabilite. Este știut că pentru efectuarea unor lucrări de mai puțină importanță sau de mai mică complexitate nu se procedează la o degrevare de sarcinile curente. Prin urmare această modalitate de conducere prin proiecte nu este avantajoasă pentru conducerea activității de programare.

B. *Conducerea pe bază de proiect cu stat major* caracterizată prin atribuirea responsabilității de coordonare unui șef de proiect și de constituirea unui colectiv care se ocupă în exclusivitate de proiect. La nevoie se poate apela și la specialiști din alte compartimente sau formații de lucru.

Dacă proiectul este amplu se poate proceda la repartizarea sarcinilor pe subcolective și organiza șefi de subcolective.

Așa cum vom vedea în paragraful următor această modalitate este avantajoasă în programare.

Pentru aplicarea eficientă a sistemului conducerii prin proiecte este necesar să se stabilească sarcinile, competențele și responsabilitățile pentru fiecare membru al colectivului. Este important ca să existe unitate de păreri și idei cu privire la metodologia de elaborare a produselor program, pentru că numai în acest fel se poate comunica în cadrul echipei de programare și realiza un control prin inspecția codului (în limbaj de programare, în pseudocod etc.).

O atenție deosebită se acordă stabilirii punctelor și modalităților de control pe parcurs și la terminarea proiectului în așa fel încît să se elimine pe cît posibil erorile în momentul producerii lor și nu mai tîrziu cînd pot genera importante cheltuieli neeconomicoase. Este bine ca șeful de proiect să aibă o experiență în elaborarea de produse-program apropiate ca nivel de complexitate și domeniu de aplicație cu cel ce urmează a fi elaborat. El trebuie să întrunească calitățile profesionale necesare manifestării în calitate de conducător.

De un real folos în procesul conducerii pot fi rezultatele unor calcule statistice prin care se pot estima costurile de elaborare a unui produs-program, intervalele de timp etc.

15.4. Organizarea echipei de programare

Conducerea pe bază de proiecte cu stat major a fost aplicată cu succes în practica programării.

Experimentată și promovată mai întîi în cadrul corporației IBM, conducerea pe bază de proiecte are corespondent în planul organizării echipei de programare, metoda cunoscută sub numele de *echipa programatorului șef*.

Această metodă are la bază stabilirea unor structuri tip de echipă de programare și delimitarea precisă a atribuțiilor fiecărui membru al echipei precum și stabilirea unor metodologii de lucru. Activitățile în cadrul unui proiect sînt organizate după principiile metodei descendente.

Conform metodei echipa programator șef pentru elaborarea unui proiect (produs-program) se constituie un colectiv de programatori format în principiu din:

A) *Un programator șef* care răspunde de coordonarea întregului proiect. El este cel care își formează, pe baza cerințelor software, o primă imagine asupra proiectului și asigură legătura permanentă cu echipa de proiectare a sistemului informatic*).

Pornind de la datele estimative cu privire la complexitatea proiectului, mărimea lui, timpul limită impus de beneficiar, resurse umane existente, necesarul de forță de muncă etc., el este cel care stabilește colectivul de elaborare și întocmește un prim plan de activitate (graficul de elaborare).

Odată colectivul de programatori constituit programatorului șef îi revin sarcini pe linia instruirii atît cu privire la specificul proiectului cît și la metodologia de lucru și respectiv de programare.

---

*) Produsul-program constituie componenta de bază a unui sistem informatic.

Împreună cu programatorul şef adjunct, şi dacă este cazul şefii pe subcolective, defineşte structura funcţională a produsului-program, stabileşte activităţile, sarcinile ce revin fiecăruia, termenele şi punctele de control. Tot programatorului şef îi revin şi sarcinile de realizare a modulelor complexe şi îndeosebi a celor de coordonare în procesul prelucrării automate a datelor.

El este cel care, împreună cu programatorul şef adjunct şi şefii de colective (dacă este cazul) realizează controlul pe parcurs în vederea validării rezultatelor intermediare şi încadrării în termenele prestabilite (eventual modificarea acestor termene).

— *Un programator şef adjunct* cu funcţii similare programatorului şef. La nevoie el va înlocui programatorul şef.

— *Unul sau mai mulţi programatori* cu sarcini pe linia dezvoltării componentelor operaţionale ale produsului-program.

— *Bibliotecarul proiectului* cu sarcini pe linia ţinerii la zi a documentaţiei proiectului (note de programare, listinguri, fişiere, colecţii de programe memorate pe suporturi, documentaţia de prezentare, de validare şi de operare etc.). El se ocupă de rularea programelor, de clasarea vechilor versiuni etc.

În funcţie de complexitatea şi dimensiunile produsului-program de elaborat se poate proceda la o organizare pe *echipe ierarhice* care să acopere întreaga activitate, plecînd de la descompunerea funcţională şi ajungînd la realizarea fiecărei componente (modul).

În practica unităţilor de informatică din ţara noastră activitatea de programare este organizată pe ateliere iar în cadrul acestora pe colective de lucru. Organizarea pe colective este realizată avînd la bază proiectele de elaborare a produselor-program incluse în planul de activitate anual.

Încercările de repartizare a sarcinilor în cadrul echipei avînd la bază componentele etapelor din ciclul de viaţă al programelor nu au dat rezultate satisfăcătoare.

Programatorii manifestă reţineri în a lucra după specificaţii de definire funcţională sau de realizare întocmite de alte persoane (alţi programatori).

Totuşi au existat şi rezultate încurajatoare atunci cînd aceste specificaţii, şi în special cele de realizare, au fost date în pseudocod gata pentru a fi codificate şi aceasta atunci cînd codificarea era realizată de programatori-ajutori care au o experienţă mai redusă în proiectarea de programe.

În ceea ce priveşte echipa de programare nu se pot încă formula păreri şi nici cuantifica exact caracteristicile care să ducă la determinarea cu exactitate a efectivului şi structurii acesteia.

Aceasta şi pentru faptul că nu se poate determina cu precizie necesarul de forţă de muncă. Normele actuale cu privire la forţa de muncă au mai mult un caracter aproximativ.

Nu se poate spune că pentru elaborarea produsului program x sînt necesare N instrucţiuni, iar pentru a realiza o linie sursă (proiecta, codifica, testa) se cere un timp de M secunde.

În practică, estimările se fac în general, prin comparaţii şi pe bază de experienţă. Ulterior se fac modificările de rigoare, în plus sau în minus.

Nici literatura de specialitate nu avansează date mai precise cu privire la productivitatea muncii, dimensiunea şi complexitatea programelor etc., indicatori care să fundamenteze ştiinţific aplicarea unui sistem de norme şi normative în programare.

Se pot da numeroase exemple de programatori care săptămîni sau luni întregi nu au scris decît cîteva linii sursă iar într-o lună au scris 1 000 linii adică exact o jumătate din norma prevăzută de către unele lucrări de specialitate pentru un programator, în timp de un an.

Ca să nu mai aducem în discuție timpul afectat punerii la punct a programelor și eventual defecțiunilor hardware care apar.

O variabilă neinițializată și folosită ulterior într-o operație pe principiul cumulării poate însemna o rulare în plus și de aici nerespectarea termenelor de predare.

Iată de ce considerăm că în aceste condiții aplicarea tehnicilor moderne de programare cum ar fi programarea structurată și eliminarea eventualelor erori cît mai aproape de momentul în care ele au fost introduse, constituie cerințe și în același timp premise pentru creșterea eficienței activității de programare.

O importanță deosebită o prezintă climatul de muncă și îndeosebi comunicarea permanentă între membrii echipei, pentru a se informa și verifica reciproc.

15.5. Indicatori fizici în programare

Conducerea în programare presupune norme, normative și indicatori care să comensureze fizic și valoric produsele program pe etape ale procesului de elaborare și pe coordonatele plan-efectiv.

La evaluarea caracteristicilor fizice se face apel la indicatori cum ar fi [74—76]:

1. *Dimensionarea vocabularului* se determină după formula

$$n = n_1 + n_2,$$

unde n_1 este numărul de operatori distincți ce apar în program (orice referință la o etichetă sau program este considerată operator); n_2 — numărul de operatori distincți care apar în program.

2. *Lungimea programului* se determină după formula

$$N = N_1 + N_2,$$

unde N_1 este numărul de utilizări a tuturor operatorilor programului; N_2 — numărul de utilizări a tuturor operatorilor programului.

Din cele două formule rezultă relația $n \leqslant N$.

Dacă se presupune că în program secvențele care se repetă se organizează sub forma unor funcții comune lungimea maximă a programului exprimată în număr de tuplete de lungime n este egală cu n^n, iar

$$N/n \leqslant n^n. \tag{15.1}$$

Cum operatorii și operanzii au tendința să alterneze în program atunci (15.1) ia forma

$$N < n\, n_1^{n_1} \times n_2^{n_2}. \tag{15.2}$$

Avînd în vedere faptul că această limită include și toate părțile care constituie programul, în număr de 2^N, obținem

$$2^N = n_1^{n_1} \times n_2^{n_2}$$

iar

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2. \qquad (15.3)$$

Studiile statistice prezentate în lucrări indică o corelare între N-ul teoretic și N-ul observat de 0,98.

3. *Volumul programului* precizează numărul minim V de biți necesari pentru implementarea unui algoritm de lungime N avînd n simboluri distincte și se notează:

$$V = N \log_2 n.$$

Studiile recente fac precizări cu privire la volumul potențial V^* dependent de limbaj stabilit după formula:

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*), \qquad (15.4)$$

pentru că $N_1 = n_1^* = 2$, $N_2 = N_2^*$ și nivel al programului L determinat

$$L = V^*/V \leqq 1,$$

iar $LV = V^*$ permite să se măsoare conținutul informațional al programului.

O aproximare a lui L se poate obține după formula

$$\widehat{L} = (2/n_1) * (n_2/N_2). \qquad (15.5)$$

Conceptele de volum și nivel ale programului s-au aplicat de către A. Champenois [75] la cele trei structuri standard de control folosite în programarea structurată. Cu această ocazie s-a putut analiza incidența cantitativă a instrucțiunilor de control dintr-un program asupra mărimilor care caracterizează complexitatea sa și durata de programare.

4. *Durata programării* se determină după formula

$$\widehat{T} = \frac{N_2 \log_2 n}{4S} \quad \text{sau} \quad \widehat{T} = \frac{n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n}{2 n_2 S}, \qquad (15.6)$$

în care valoarea lui n se obține din formula

$$N = n \log_2 (n/2).$$

Dezavantajul folosirii acestei formule constă în faptul că ea nu se poate utiliza în faza de planificare întrucît nu este posibil să se evalueze n_1, n_2, N_1 și N_2.

Pentru stabilirea formulei de determinare a duratei programării s-au făcut următoarele ipoteze:

— orice programare al unui algoritm constă în a ordona N elemente (operatori sau operanzi) alegînd într-un vocabular format din n elemente;

— selecția fiecăruia din cele N elemente se face urmînd o cercetare dicotomică a cărei lungime medie este $\log_2 N$;

— pentru realiza ordonarea celor N elemente ale programului, numărul total de comparări mintale (dicotomice) este $N \log_2 n$;

— acest număr de comparări mintale apare ca fiind volumul programului $V = N \log_2 n$;

— fiecare comparare mentală presupune un efort proporțional cu dificultatea acestei sarcini; efortul de programare este invers proporțional cu L;

— numărul total E de discriminări mentale elementare se presupune a fi: $E = V/L = V^2/V^*$;

— viteza medie de discriminare S se situează între 5 și 20 biți pe secundă

5. *Nivelul limbajului* este determinat după formula:

$$\lambda = LV^* = L^2V$$

și prezintă interes pentru estimarea duratei previzionale de programare și pentru a compara productivitatea folosind mai multe limbaje.

În aceste condiții E se mai poate calcula după formula

$$E = V/L = V^{*3}/\lambda^2,$$

unde V^* depinde practic de numărul operațiilor de intrare/ieșire ale programului de construit.

Pentru a calcula durata programării se va porni de la n_2^* numărul operațiilor de intrare și de ieșire din program și nivelul limbajului folosit.

În continuare se va calcula V^* după formula

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$$

și E după formula

$$E = (V^*)^3/2.$$

Presupunînd că dispunem de o definire precisă și completă a algoritmului de programat și că programatorul are o experiență în folosirea unui anumit limbaj

$$T = E/S \text{ unde } (S \approx 18/S).$$

Formulele indicatorilor fizici precizați se pot aplica și pentru alte evaluări cum ar fi: detectarea erorilor și modularitatea.

Întrucît cercetările actuale nu sînt încă în măsură să furnizeze formule fundamentate științific ne limităm doar la a afirma că realizarea conducerii științifice a activității de programare nu va fi posibilă decît în condițiile în care se va preciza sistemul atributelor fizice și evalua pe baza acestora indicatorii fizici în programare.

15.6. Factorii care influențează costul programelor

Desfășurarea pe principii economice a întregii activități care are ca obiectiv elaborarea programelor aduce în prim plan necesitatea calculației costurilor și prin urmare studierea factorilor care influențează costurile programelor.

Pe această bază se pot lua decizii importante cu privire la alegerea unui produs-program; organizarea echipei de programare și în general conducerea activității de elaborare a produselor-program.

În [76] B. W. Boehm în baza unei sinteze reușite a cercetărilor cu privire la costul programelor împarte în cinci clase factorii care determină costurile și anume:

1. *caracteristicile de mărime* măsurate în număr de instrucțiuni, număr de date, de intrări sau de ieșiri;

2. *caracteristicile produsului-program* care privesc fiabilitatea, complexitatea, limbajul utilizat etc.;

3. *caracteristicile personalului* care privesc experiența acumulată în domeniul programării, organizarea echipei de programare, calificarea personalului etc.;

4. *caracteristicile calculatorului* care privesc limitele vitezei de execuție sau memoria ocupată etc.;

5. *caracteristicile proiectului* care privesc instrumentele utilizate, metodele folosite în programare, limitele cu privire la termenele de predare etc.

Mărimea programului măsurată în număr de Linii Cod instrucțiuni sursă Livrate — LCS — (ISL în notația autorului) permite o evaluare rapidă a costului produsului program.

La evaluarea indicatorului LCS se au în vedere: liniile sursă ale programului principal cu instrucțiunile de prelucrare și de descriere a datelor; liniile cu instrucțiunile adresate limbajului de comandă; liniile programelor care nu sînt livrate clientului și care sînt construite ca instrumente de testare.

Notă. Dacă mai multe linii sînt scrise pe o linie — imagine de cartelă — ele intră în calcul ca fiind o singură linie.

Desigur că se poate opta și pentru măsurarea mărimii programului în număr de module, număr de intrări, număr de ieșiri, sau număr de linii în limbajul de specificare a problemei dar aceste măsuri sînt globale și nu permit decît estimări tot globale cu privire la costuri.

Indicatorul fizic „volumul programului" propus de Halistead [77] pare să coreleze mai bine mărimea programului și costul de elaborare și să pună astfel bazele unor studii statistico-experimentale în vederea elaborării unor norme unitare.

Într-un studiu făcut asupra unui program operațional de 100 000 LCS B. W. Boehm [78] constată că numărul liniilor cod-sursă livrate beneficiarului poate să fie cuprins între 100 000 și 500 000 LCS.

Pe baza aceluiași studiu s-a ajuns la concluzia că o creștere cu o unitate a mărimii programului determină o creștere a costurilor cu mai mult de o unitate.

Se estimează că relația între efortul exprimat în om × luni (OL) și mărimea programului (MISL adică mii linii sursă livrate) este în general de tipul: OL = A · MLCS. B. Cînd $B > 1$, se ajunge la un randament descrescător.

Numărul liniilor sursă livrabile prezintă importanță deosebită pentru măsurarea productivității muncii exprimată în LCS/om-luni.

Caracteristicile cum ar fi fiabilitatea, tratate pe larg în capitolul 11, influențează costurile în sensul că pentru a obține un produs-program cu fiabilitate foarte ridicată se vor face cheltuieli mai mari.

Printre caracteristicile produsului-program o importanță deosebită o are mărimea fisierelor și a bazei de date ce urmează să se proiecteze și valideze care determină o mărime a numărului liniilor sursă și implicit a costului de realizare a programului.

Complexitatea programului influențează și ea costurile în sensul că, cu cît un program este mai complex, cu atît și costul de elaborare va fi mai mare; considerînd că ceilalți factori rămîn constanți.

Deși studiile recente consideră productivitatea în programare (exprimată în LCS) independentă ea este independentă de nivelul limbajului (cu precizarea ca aceasta să nu fie decît în cazuri extreme un limbaj de asamblare).

Folosirea limbajelor evoluate permite o reducere considerabilă a costurilor, mai ales a celor de testare, depănare și întreținere a programelor.

Desigur că dacă avem în vedere posibilitățile oferite de limbajele evoluate și cerințele software privind produsul-program, alegerea limbajului din mulțimea limbajelor evoluate are influență apreciabilă asupra productivității muncii și respectiv asupra costurilor de realizare a produsului-program.

În [2, 94, 96, 79] prin care s-au comparat mai multe proiecte după nivelul limbajului de programare folosit se estimează o economie de la 25 la 37% pe ansamblul ciclului de viață al unui program și creșterea productivității muncii între 33 și 59%.

Cîștiguri și mai mari se înregistrează în faza de actualizare (menținere), astfel se apreciază că în funcție de nivelul limbajului personalul necesar pentru menținerea aceluiași număr de instrucțiuni poate varia într-un raport de 1 la 4.

În ceea ce privește productivitatea în programare, aceasta exprimă, într-o formă sintetică, influența a o serie de factori cum ar fi: competența celor care elaborează programul; experiența în folosirea unui anumit limbaj de programare; experiența în folosirea unor instrumente software de ajutor în programare; exigența cu privire la portabilitatea programelor; folosirea metodelor moderne de programare; complexitatea problemei și a programului de realizat, instrumentele software de dezvoltare a programelor și îndeosebi cele de gestionare a codului; organizarea echipei de programare etc.

Se înțelege că nu toți factorii influențează în aceeași măsură productivitatea muncii. Studii statistico-experimentale pot permite stabilirea unor rate de productivitate pe diferiți factori procedînd după metoda substituției cunoscută în analiza economică.

Caracteristicile calculatorului (timpul de execuție, mărimea memoriei etc.) și ale sistemului de operare influențează și ele costurile de dezvoltare a programelor. Experiența autorilor pe linia dezvoltării în regim iteractiv a unui program complex de asistență în învățămînt a permis să se constate reduceri considerabile a timpului care a ajuns pînă la 55%.

Caracteristicile personalului, au cea mai mare influență asupra costurilor. Cu cît personalul este mai bine pregătit cu atît productivitatea muncii crește iar costurile scad.

Menționăm printre caracteristicile personalului: nivelul de calificare; experiența practică acumulată în programare; experiența practică acumulată în elaborarea de programe în domeniul de aplicație similar; experiența practică acumulată în lucru cu sistemul de calcul pe care urmează să fie elaborat programul; experiența practică în dezvoltarea de programe folosit limbajul de programare ales; calitatea organizării echipei de programare.

Calificarea personalului trebuie analizată la nivel individ dar mai ales la nivel echipă de programare, știut fiind că munca în echipă presupune coeziune și comunicare permanentă între programatori.

Referitor la utilizarea metodelor de inginerie a programării, numeroase studii au scos în evidență o reducere a costurilor, îndeosebi a celor de gestionare cod, integrare, testare și întreținere (pînă la 40% față de costurile realizate în condițiile folosirii metodelor clasice).

În [23] se estimează ca efecte previzibile ale utilizării metodelor de inginerii a programării (vezi și tabelul 15.1): creșterea considerabilă a productivității muncii în programare (de la 10 ÷ 15 instrucțiuni corecte pe zi la 20 ÷ 25 și chiar 100 instrucțiuni corecte pe zi); creșterea fiabilității (de la 1 ÷ 5 erori la 100 instrucțiuni la 1÷5 erori la 10 000 instrucțiuni); scăderea costului întreținerii cu 10 și chiar 20%.

Tabelul 15.1. Efectele utilizării metodelor de inginerie a programării

Faza	% din costul total în prezent	% reducere previzionată a costului	% din costul total în viitor
Specificare cerute	10	20	8
Proiectare generală	15	33,3	10
Proiectare de detaliu	20	40	12
Codificare	10	75	4
Testare componente	10	50	5
Integrare și testare de ansamblu	30	25	23
Alte activități	—	—	37
TOTAL	100	circa 40	100

Analizele efectuate citează creșteri considerabile ale productivității muncii în condițiile folosirii metodelor de inginerie a programării care se situează între 44 și 58%. Facem, însă, precizarea că în aceste studii nu se delimitează strict care este creșterea datorată folosirii metodelor de inginerie a programării și care este creșterea cauzată de folosirea unor instrumente de ajutor în programare. Se menționează doar faptul că trecerea de la un mediu de programare foarte bogat în instrumentele software de ajutor la un alt mediu sărac în astfel de instrumente atrage după sine scăderi cu peste 30% a productivității muncii. Menționăm din experiența Centrului de calcul din cadrul CNST care dezvoltînd program COBOL pe sistemul WANG folosind instrumentul COBGEN care a înregistrat creșteri de productivitate cu peste 40%.

Fixarea unor limite cu privire la termenul de elaborare a produsului-program poate, de asemenea, influența costul acestuia. Spre exemplu, scurtarea pe parcurs a termenelor prin adăugarea de personal poate uneori influența negativ calitatea produsului-program.

În [94] se precizează că există pentru orice proiect o limită minimă peste care nu se poate reduce intervalul de elaborare a unui produs-program și care este situată în jurul a 75% din durata nominală determinată folosind formula

$$\text{Durata nominală} = 2{,}5\sqrt[3]{OL}$$

unde OL măsoară efortul necesar.

15.7. Sistemul de norme și indicatori

Întocmirea planului economico-social, care reprezintă programul întregii activități al unităților de informatică, urmărirea modului de îndeplinire a planului și stabilirea rezultatelor activității pe sectoare și locuri de muncă presupune existența unui sistem unitar de norme și indicatori pentru corelarea direcțiilor tehnice, economice, financiare și sociale [72].

La stabilirea sistemului de norme și indicatori pentru activitatea de programare se pleacă, cum este și firesc, de la faptul că produsele-program au o valoare dată de cheltuielile medii socialmente necesare pentru elaborarea produsului și o valoare de întrebuințare dată de utilitatea acestora.

În domeniul programării normele stabilesc cheltuielile maxime admise de munca vie și materializată necesare elaborării de programe. Pentru ca normele să aibă valoare aplicativă ele trebuie să rezulte pe bază de măsurători statistice pe faze ale elaborării produselor-program și pe tipuri de programe.

Normele unitare trebuie să fie în același timp progresive și dinamice.

Considerăm necesar să se includă în sistemul normelor aplicate la activitatea de programare: normele de folosire a muncii vii; normele de folosire a sistemului de echipamente; normele de folosire a obiectelor muncii; norme de durată a ciclului de elaborare a produselor-program.

La stabilirea normelor și normativelor unitare pe economie națională este necesar să se facă distincție între: produse-program cu aplicație în domeniul economic; produse-program cu aplicație în domeniul științific; produse-program de conducere a proceselor tehnologice; produse-program de gestiune a bazelor de date; produse-program pentru lucrări în regim de teleprelucrare; produse-program, componente ale sistemelor de operare etc.

Pentru aceste categorii de programe se va lua în calcul și sistemul standardelor prevăzute pe fiecare fază de elaborare;

Cum elaborarea planului economico-social are în vedere și celelalte laturi ale activității desfășurate în unitățile de elaborare a produselor-program și îndeosebi în activitatea financiară, o importanță deosebită o prezintă evaluarea indicatorilor naturali și valorici dintre care enumerăm: numărul de instrucțiuni pe produse-program; numărul produselor-program pe tipuri de produse-program și grad de complexitate; numărul de programe pe tipuri de produse-programe; costurile de realizare a produselor-program; fondul de retribuire; producția marfă; producția netă etc.

Fără a intra în detalii apreciem că particularitățile activității de programare fac dificilă satisfacerea necesităților de planificare a indicatorilor cuprinși în plan grupate în două mari categorii:

— *indicatori cantitativi* prin care se dimensionează volumul producției în expresie naturală și valorică, efectivul de personal pe specialități, fondul de retribuire a muncii etc.;

— *indicatori calitativi* prin care se apreciază calitatea muncii și a produselor program în care sînt incluși indicatorii tehnici și economici cum ar fi: consumurile specifice; costurile pe produs-program, productivitatea muncii etc.

Sistemul normelor și indicatorilor stă la baza întocmirii programului de „producție" folosind indicatori naturali, valorici și unități de timp de muncă.

În cadrul indicatorilor naturali care servesc la stabilirea programului de activitate privind programarea un loc central îl ocupă indicatorii în unități de timp de muncă cum ar fi spre exemplu *ore-om*.Acesta exprimă timpul în ore necesar pentru elaborarea produsului-program cu defalcare pe etape ale procesului de programare.

Cu ajutorul indicatorilor valorici se determină volumul „producției".

În ideea organizării unor întreprinderi specializate în producerea de produse-program de uz general, produsul-program constituie și produsul finit, care este evaluat cu ajutorul indicatorilor valorici.

Referitor la normele de muncă cu privire la activitățile specifice de realizare a sistemelor informatice [79] acestea prevăd la paragraful 1.3 cu privire la elaborarea programelor, norme distincte pentru analiști și programatori.

Repartizarea în procente a normelor de muncă a activităților ce urmează a fi executate de programatori se face pe structura: proiectarea produsului program și întocmirea documentației (35% din norma de muncă); dezvoltarea programelor (50% din norma de muncă); conducerea, organizarea, coordonarea și controlul pe durata proiectării și dezvoltării (10% din norma de muncă); activității de avizare-recepție pe parcursul și la sfîrșitul elaborării produsului program (10% din norma de muncă); activității de editare a documentației (3% din norma de muncă).

Elementele de bază folosite în calculul normei de muncă pentru activități de programare sînt: tipul funcției; complexitatea procedurii de prelucrare a datelor; numărul de fișiere; diversificarea datelor.

Spre exemplu, pentru realizarea unei proceduri normale de muncă — conform metodologiei actuale — se determină după formula

$$N_j = 8 \cdot [D \cdot (C + 2) + F \cdot P/3],$$

unde D este coeficient de diversitate a datelor ce se calculează la rîndul lui, astfel

$$D = \sum_{i=1} n_i k_i,$$

unde k_i este un coeficient; n_i — numărul de înregistrări logice, entități sau liste; C — complexitatea procedurii; P — volumul și dificultatea prelucrării; F — numărul de fișiere.

Experiența aplicării în practică a acestor norme dovedește că ele sînt suficient de largi. Totodată ele prezintă și limite prin faptul că nu fac referiri cu privire la normele de elaborare a programelor de gestionare a bazelor de date și respectiv de lucru în timp real și în general diferențieri cu privire la categoriile de produse program.

Aceasta presupune studii și experimentări și în primul rînd o vastă experiență în acest domeniu.

Considerăm că pe baza normelor de muncă a fondului de timp necesar pentru îndeplinirea programului de „producție" și a fondului efectiv al fiecărui specialist se poate determina necesarul de forță de muncă. Valorile rezultate sînt supuse unor calcule care vizează îndeosebi dimensiunea optimă a echipei de programare pentru a determina astfel și timpul calendaristic necesar pentru elaborarea unui produs-program.

Folosind același sistem al normelor se poate proceda la calculația costului de deviz în structura pe articole de calculație specifică acestui domeniu de activitate.

Pentru aprecierea eficienței activității de programare se vor dimensiona [73]: complexitatea problemei; complexitatea fiecărui program; consumul de muncă pentru elaborarea fiecărui produs-program; complexitatea netă a fiecărui program (în care nu se include complexitatea componentelor încorporate); consumul de resurse materiale pentru realizarea fiecărui program; consumurile de resurse specifice unui ciclu de exploatare pentru fiecare program; valoarea fiecărui program.

În baza acestor indicatori considerăm că se poate stabili productivitatea muncii în programare, raportul dintre numărul de instrucțiuni și numărul de operații convenționale; raportul între valoare și numărul de operații, raportul între valoare și numărul de instrucțiuni; raportul între costul exploatării și numărul de operații etc.

Stabilirea unui sistem unitar de norme, normative și indicatori constituie premisa organizării după principii economice a activității de programare și instituirea unui regim economic al produselor-program.

Dintre indicatorii calitativi un loc important îl ocupă productivitatea muncii. Cu privire la productivitatea muncii au fost păreri care au susținut că ea se dovedește a fi mai corectă la nivel macro decît la nivelul programatorului individual [97].

În paragraful 1 au fost analizate diferite posibilități de calculare a productivității muncii în programare.

Vom prezenta în continuare posibilitățile de determinare practică a productivității muncii.

Să luăm spre exemplu numărul de linii cod sursă (LCS) pentru o persoană într-un interval de timp considerat (de exemplu *o oră, zi sau lună*).

Aplicarea în practică a unui astfel de indicator presupune o corelare strînsă cu complexitatea programului.

Așa se explică faptul că referințele la productivitatea muncii citează de la 1 LCS/oră pînă la 30 LCS/oră.

În [84, 98] se face gruparea pe nivele de complexitate a programelor astfel:

Nivel	*Tipul de program*	*Multiplicator de cost*
1	program	(în care este inclusă productivitatea muncii) 1X
2	produs-program	3X
3	sistem de programare	3X
4	produs-sistem de programare	9X

Termenul om/zi folosit în determinarea productivității muncii este definit ca fiind timpul în zile necesar pentru îndeplinirea unei sarcini și se poate estima pe faze ale procesului de elaborare a programelor.

Se apreciază că în programare timpul productiv reprezintă între 50÷80% din timpul total. Prin urmare putem calcula productivitatea muncii pentru om/zile total, și om/zile productiv.

Ponderea mare a timpului neproductiv este dată de specificul activității de programare.

La estimarea numărului liniilor de cod sursă LCS se pornește de la ideea că o instrucțiune reprezintă o linie sursă.

Spre exemplu, pentru programe realizate în limbajul COBOL se apreciază ca linii cod sursă: declarațiile din deviziunile INDENTIFICATION, ENVIROMMENT și DATA; verbele din diviziunea PROCEDURE (care exprimă cel mai bine complexitatea programului); liniile cu comentarii; instrucțiunile de comandă.

Pentru a asigura comparabilitate în ceea ce privește indicatorul productivitatea muncii este necesar să se precizeze clar ipotezele de calcul (ceea ce în practică este greu de realizat întrucît în programare nu dispunem încă de un sistem unitar de standarde).

Variația productivității individuale este strîns legată de definirea liniei de Cod. Iată de ce considerăm mai indicat pentru a aigura comparabilitatea la nivel macro, ca orice linie sursă din program să fie apreciată ca fiind o linie de cod sursă.

Numărul liniilor de cod sursă (determinat îndeosebi pe baza verbelor din procedură) se poate folosi în munca de planificare și control pentru a determina dimensiunea și complexitatea programului.

Cum munca de programare este colectivă este necesar să se calculeze productivitatea la nivel proiect calculată ca raport între numărul de linii cod sursă realizate de către întreaga echipă și numărul de om/zile afectate.

Așa cum se arată în [97] pentru a se asigura comparabilitate indicatorului productivitatea muncii la nivel proiect este necesar să existe o bază de referință comună.

Între proiecte pot apărea diferențe îndeosebi în legătură cu: complexitatea conducerii proiectului și detalierea specificațiilor.

Productivitatea muncii la nivel proiect este influențată direct de următorii factori principali: calificarea personalului, metoda și tehnicile folosite în programare și mediul de programare.

Referitor la tehnicile și mediile folosite în programare în [97] se fac următoarele estimări cu privire la procentul de creștere a productivității în etapa de implementare*) (tabelul 15.2).

Tabelul 15.2. Estimări de creștere a productivității în etapa de implementare

Nr. crt.	Tehnică	Procent %	Nr. crt.	Tehnică	Procent %
1	Programare structurată	10	8	Diagrame de structură	10
2	HIPO	0	9	Generator de rapoarte	15
3	Proiectare descendentă	20	10	Compilare și depănare om-linie	10
4	Programare structurată	5	11	Module standard	30
5	Organizarea echipei	5	12	Generator de aplicații	înalt
6	Verificări structurate	3	13	Cod autogenerator	înalt
7	Bibliotecar manual om-linie	5			

Productivitatea muncii ca principal indicator de măsurare a eficienței în programare, îndeosebi exprimată la nivel proiect, trebuie folosită în complexul proces de planificare, urmărire și control.

Pentru aceasta trebuie însă să existe o bază de referință fundamentată științific, ceea ce aparține de domeniul viitorului în programare.

15.8. Tipizarea și standardizarea—premise ale creșterii eficienței economice în elaborarea produselor-program

Extinderea domeniilor de utilizare a calculatoarelor electronice a făcut ca în practică, în același timp, programatorii din unități diferite specializate în prestarea de servicii în domeniul informaticii, să aibă ca sarcină pe linia elaborării de produse program probleme similare atît ca funcții și domenii de aplicații cît și ca parametrii și efecte externe.

*) În accepțiunea de codificare, inserare, test, sistem, documentare și conversii.

În aceste condiții trecerea la elaborarea unor *produse-program tip* și *standardizarea* acestora constituie o cerință pe linia eficientizării activității de informatică și în același timp premisa introducerii unor norme, normative și indicatori fundamentați științific care să asigure planificarea, urmărirea și controlul utilizării resurselor în activitatea de elaborare a produselor-program.

Cadru organizatoric și normativ unitar permite proiectarea unor sisteme informatice tip, pe domenii de aplicație și implicit a unor produse-program tip.

Cerințele sistem și respectiv cerințele software trebuie elaborate avînd în vedere și obținerea unor produse-program tip pe domenii de aplicație, utilizabile de către un număr tot mai mare de beneficiari.

Standardizarea ca sistem de măsuri privind determinarea și introducerea unor mărimi, însușiri și calități obligatorii pentru produsele-program, constituie o importantă pîrghie de plan pentru ridicarea nivelului tehnic și calitativ al producției [72].

Tipizarea și standardizarea permit desfășurarea unor acțiuni pe linia reglementării mărimilor, însușirilor și calității produselor-program și definirii unor norme unitare privind resursele necesare în vederea elaborării lor.

Surprinde faptul că deși există o experiență apreciabilă în elaborarea unor produse-program, deși există anumite standarde pe etape ale procesului de elaborare a programelor pentru diverse aplicații, în planificarea și urmărirea resurselor se folosesc încă norme globale pentru determinarea cheltuielilor în vederea încheierii contractului cu beneficiarii.

Considerăm că ar trebui să existe norme unitare pe tipuri de produse-program și pe faze ale elaborării acestora care să aibă la bază în primul rînd standardele existente.

Se pot stabili norme medii pe bază de studii științifice luînd în calcul: strategiile și variantele de rezolvare a problemelor cu ajutorul calculatoarelor electronice; obiectivele stabilite de utilizatori; configurația de calcul pe care va fi exploatat produsul-program; limbajul folosit în programare etc.

În sistemul normelor se vor cuprinde norme de personal, bazate pe indicatorii cu privire la productivitatea muncii și dimensiunea produsului-program, norme de cheltuieli etc.

Numai în felul acesta se poate ajunge la obținerea unor proceduri, programe, produse-program etc. normalizate pe domenii de activitate adaptabile cu ușurință unor modificări ce intervin pe parcurs.

Perfecționările aduse în plan tehnologic și metodologic, trecerea de la intuiție și experiență la știință și industrie în programare permit elaborarea unor proceduri și produse-program tip, pe domenii de activitate.

Tipizarea și standardizarea vor duce implicit la constituirea pe plan național a unei colecții de proceduri și produse-program tipizate, la standarde și norme unitare ceea ce va însemna o creștere considerabilă a eficienței economice a activității de informatică.

Produsele-program vor fi preluate din biblioteci de programe standard (create pe ramuri de activitate și probleme tipizate) și adoptate propriilor nevoi cu cheltuieli minime. Din cuprinsul acestei lucrări a rezultat că se pot elabora produse-program de uz general autoadaptabile la cerințele fiecărui utilizator (este cazul generatoarelor de produse-program).

Conducerea activității de programare trebuie făcută în conformitate cu cerințele obținerii unor produse-program de uz general tipizate și de calitate superioară.

Elaborarea de produse-program tip pe domenii de activitate și în cadrul acestora pe aplicații deschide perspectiva specializării și organizării într-o nouă concepție a unităților de informatică.

Se poate avansa ideea organizării unor unități specializate, ateliere, în proiectarea și realizarea unor categorii de produse-program.

În aceste condiții multe din actualele centre de calcul vor avea cu precădere responsabilități pe linia implementării și exploatării produselor-program. Ele pot eventual asigura și operațiile de adaptare și întreținere pentru domeniul în care s-au specializat, realizînd astfel legătura între proiectant și beneficiar.

15.9. Organizarea controlului de calitate

Controlul de calitate este un proces complex care trebuie să se regăsească pe toate etapele procesului de elaborare a programelor. În scopul efectuării lui trebuie precizate punctele de control, stabilit necesarul de resurse și indicat cadrul metodologic de efectuare. Controlul de calitate trebuie astfel organizat încît să asigure depistarea la timp a erorilor, satisfacerea cerințelor specificate de către utilizator, asigurarea livrării la termen și reducerea costului produsului informatic în condițiile creșterii parametrilor funcționali și diminuarea la maxim a erorilor ce se depistează în faza de exploatare.

Se știe că a corecta o eroare în momentul în care produsul-program este operațional înseamnă a cheltui de la 10 la 100 ori mai mult decît dacă ea ar fi fost depistată și înlăturată cît mai aproape de momentul introducerii ei.

După cum se arată în [85, 92] procesul de control al calității produselor-program se realizează prin următoarele activități de bază:

a. *Verificare*, care are ca obiectiv să verifice dacă rezultatul obținut la fiecare etapă a ciclului de elaborare a unui produs-program satisface specificațiile definite în etapele anterioare.

b. *Validarea*, care are ca obiectiv să certifice că cerințele funcționale, tehnice și economice definite pentru produsul-program sînt satisfăcute după fiecare etapă a ciclului de elaborare a unui produs-program și că produsul-program obținut funcționează în condițiile specificate la început.

c. *Certificare*, care are ca obiectiv să certifice calitatea unui produs-program pe baza rezultatelor puse la dispoziție de către o *comisie tehnică de control* care a efectuat verificarea și validarea produsului-program pe parcursul procesului de elaborare a acestuia.

Se subînțelege că dacă în procesul de control se detectează erori, este necesară înlăturarea lor iar după aceea soluția din nou supusă controlului de calitate.

În final, după ce produsul-program a fost elaborat se realizează *certificarea calității*.

Pentru efectuarea controlului de calitate în [85] se propune organizarea punctelor de control a calității produselor-program ca în figura 15.1.

La definirea procesului de control se urmărește să se răspundă la întrebările: ce se controlează?; cînd se controlează?; cum se controlează?; de ce se controlează?.

În [84, 85] se prezintă în detaliu organizarea procesului de verificare și validare pentru produse-program generalizabile pe următoarele etape: elaborarea temei de realizare; proiectarea logică de detaliu; proiectarea tehnică de detaliu.

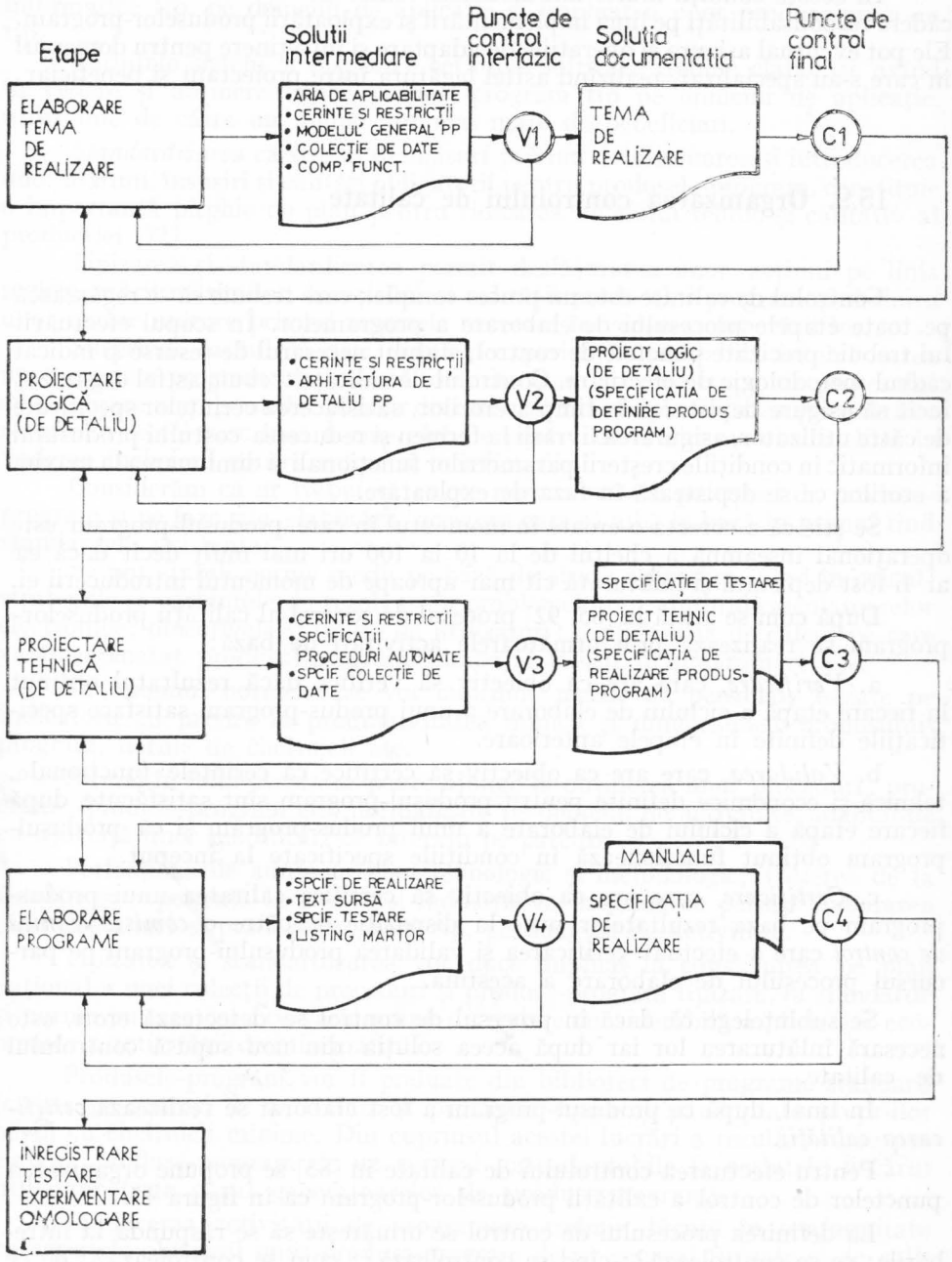


Fig. 15.1. — Puncte de control in procesul de realizare a produselor-program.

Spre exemplu pentru proiectarea logică de detaliu este necesar să se precizeze: soluția supusă verificării; componentele soluției (cerințe și restricții de detaliu și arhitectura de detaliu a produsului-program); puncte de control interfazic; punctul de control final; grila de verificare (pentru exemplificare, vezi figura 15.2); lista de verificări.

ATRIBUTE DE CALITATE	CERINȚE ȘI RESTRICȚII DE DETALIU	Modelul de detaliu al produsului-program				
		COMPONENTE FUNCȚIONALE	COLECȚIE DE DATE	INTERFEȚE EXTERNE	STRUCTURA PRODUSULUI-PROGRAM	CONȚINUTUL PROIECTULUI LOGIC DE DETALIU
Consistenta externă	X	X	X	X	X	X
Completitudine	X	X	X	X	X	X
Acuratețe	X	X				X
Securitate			X			X
Conciziune			X			X
Claritate	X	X	X	X	X	X
Consistență internă	X	X	X	X	X	X
Simplitate			X	X	X	X
Stabilitate			X		X	X
Autodescriere		X	X	X	X	X
Extensibilitate			X		X	X
Generalitate		X	X			X
Modularitate			X		X	X

Fig. 15.2. — Grila de verificare pentru etapa proiectarea logică de detaliu.

Să menționăm că organizarea controlului de calitate este în curs de a găsi soluții practice capabile să asigure obținerea unor produse-program performanțe.

Organizarea controlului calității produselor-program constituie principala cale de creștere a calității produselor-program.

Strîns legată de controlul de calitate este și problema *standardelor* *) *de asigurare* și control a calității.

Dată fiind unitatea și diversitatea standardelor, D. Ross [124] emite conceptele de META-STANDARD, meta-standardele avînd ca obiectiv [123] limitarea posibilităților de opțiune prin faptul că el stabilește un cadru de acțiune general pe baza căruia pot fi elaborate sau adoptate standarde locale la nivel de grupuri de organizații, organizații sau chiar de proiect; totodată, stipulează lista minimală a standardelor care trebuie elaborate și aplicate.

*) Prin standarde se înțelege ansamblul de reguli de specificare, reprezentare și evaluare a soluțiilor în procesul de evaluare a software-ului.

Un exemplu de meta-standard este cel intitulat "Standard for Software Anality Assurance Plans", elaborat de Software Engineering Subcomitee din cadrul IEEE Computer Society [125] și destinat elaborării și întreținerii produselor software critice, pentru care lipsa de fiabilitate poate cauza pierderi importante. Precizăm că un subset al acestui standard poate fi aplicat pentru orice tip de produs software.

Standardele susmenționate precizează conținutul *„Planului de asigurare a calității"* document întocmit obligatoriu de elaborator împreună cu beneficiarul în momentul declanșării procesului de elaborare a unui produs software și urmărit cu strictețe pe parcursul elaborării acestuia.

Literatura de specialitate [125, 123] menționează că acest plan constituie documentul prin care se stabilește tehnologia produsului și conține în principal:

— denumirea produsului și domeniul de aplicație;

— lista sarcinilor, secvența de realizare în timp, responsabilitățile și executanții, moduri de raportare a realizării sarcinilor;

— documentația care se întocmește pentru realizarea produsului și criteriile de verificare a calității;

— standardele, practicile și convențiile care se vor utiliza și eventualele concilieri admise;

— instrumentele, tehnicile și metodologiile care se vor utiliza;

— lista verificărilor și controalelor tehnice care se vor efectua pe parcursul realizării produsului;

— convenții privind identificarea produsului și a componentelor sale, regulile de control al modificărilor apărute;

— metode și facilități pentru întreținerea și înregistrarea versiunilor succesive ale produsului, biblioteci de programe;

— metode și facilități pentru protejarea fizică a suporților tehnici, a programelor și documentațiilor față de accesul neautorizat sau degradări;

Se menționează că aplicarea acestor standarde într-o societate producătoare de software din Japonia a permis obținerea unor rezultate remarcabile: creșterea anuală a productivității muncii în perioada 1976—1980 cu 14% anual; modificarea ponderii din efortul total pe etape ale ciclului de viață în favoarea fazelor de concepție software cu 10%; creșterea substanțială a numărului de componente software reutilizabile.

Aceste succese pledează pentru generalizarea folosirii unor asemenea standarde în condițiile organizării pe principii industriale a elaborării produselor software în țara noastră.

Dacă importanța existenței unor standarde este pentru toți evidentă, rezolvarea unei asemenea probleme pe plan național, trebuie bine gîndită.

Sugestivă ni se pare în acest sens încheierea făcută în [129] cu privire la principiile enunțate de S. T. Ross care vor trebui avute în vedere în orice demers de standardizare:

— standardele PROPUSE pot fi foarte folositoare;

— standardele IMPUSE pot fi dezastruoase;

— standardele sînt nu numai fără sens, ci chiar periculoase dacă utilizarea lor corectă nu este înțeleasă de cei afectați de aceste standarde;

— cea mai importantă clauză a unui standard este CLAUZA DE APLICABILITATE, care specifică exact cazurile în care NU se utilizează;

— metodele standardizate pot deveni standarde de metode numai dacă sînt ACCEPTATE în utilizare;

— COMPATIBILITATEA poate fi mai ușor dobîndită decît CONFORMITATEA, iar în mod uzual aceasta este ceea ce ne trebuie;

— standardele există numai pentru a SERVI NECESITĂȚILE unei lumi nestandard; standardele nu pot crea o lume standard.

După părerea autorilor aceste afirmații pot fi reanalizate în corelare cu mediile de programare tratate în capitolul 17.

15.10. Conducerea prin costuri

Așezarea pe baze economice a întregii activități de programare aduce în prim planul preocupărilor conducerea prin costuri.

Pentru fiecare loc de activitate și respectiv produs-program trebuie planificate, evidențiate și controlate cheltuielile astfel încît să se ajungă în final la obținerea de produse-program cu cheltuieli cît mai reduse.

Informația costurilor poate motiva importante decizii cu privire la îmbunătățirea tehnologiei de „fabricație" a programelor, a metodologiei folosite și îndeosebi perfecționarea și extinderea instrumentelor de ajutor în programare.

Pentru fiecare produs-program este necesar ca la obținerea lui în forma livrabilă să se cunoască: costul normat de elaborare; abaterile față de costul normat, cauzele producerii acestora și responsabilitățile implicate; costul efectiv de elaborare; costul exploatării produsului-program.

Costul elaborării produsului-program (normat și, respectiv, efectiv) însumează cheltuielile (normate și, respectiv, efective) pe etape ale procesului de elaborare a programelor după formula generală

$$C = \sum_{a=1}^{t} \left(\sum_{d=1}^{n} CH_{D_d} + \sum_{i=1}^{m} CH_{I_i} \right) a,$$

unde C este costul elaborării unui produs-program, a sînt atelierele care participă la elaborarea unui produs-program (în cazul în care sînt organizate ateliere pe principalele faze ale elaborării programelor), CH_D — cheltuieli ce se pot localiza direct pe produs-program, CH_I — cheltuieli indirecte, d și i, articole de calculație.

Pentru dimensionarea cheltuielilor directe și respectiv indirecte se pot folosi procedeele specifice metodelor standard cost.

Costul exploatării, care după părerea noastră prezintă interes îndeosebi pentru beneficiar, însumează cheltuielile generale de folosire a resurselor unui sistem de calcul în procesul de execuție.

În continuare prezentăm un model de contabilizare a resurselor utilizate de către program, bazat în general pe timpul de execuție a programului și memoria ocupată.

Se consideră că un cost este cu atît mai real cu cît el reflectă fidel resursele efectiv utilizate în procesul de prelucrare. Determinarea costului de utilizare a calculatorului se poate face aplicînd tarife unitare pe resurse folosite. În acest scop se cer cunoscute ratele de recuperare a cheltuielilor efectuate pentru fiecare resursă (dispozitiv) în parte.

La stabilirea ratelor de recuperare (sau a costurilor normate) se au în vedere cheltuielile generate de fiecare dispozitiv în parte și durata de funcționare.

Ratele se pot estima cunoscînd: prețul de achiziționare a diverselor componente; cheltuielile pentru materiale; fiabilitatea elementelor din configurație (timpul mediu de funcționare pînă la prima pană) și costul mediu al diverselor reparații; timpul mediu de viață al componentelor din configurație.

De exemplu, fie dispozitivul d_i pentru care avem: prețul p_i; timpul mediu de funcționare pînă la prima pană, f_i (dat prin documentație); costul mediu al unei reparații, r_i; timpul mediu de viață, v_i (dat prin documentație).

Cheltuielile pe perioada T, pentru dispozitivul d_i, sînt date de formula

$$Cd_i = p_i + (Q(v_i/f_i) \cdot Q(T/v_i) + Q(R(T/v_i)/f_i)) \cdot r_i + Q(T/v_i) \cdot p_i, \qquad (15.7)$$

unde Q și R reprezintă partea întreagă a cîtului și respectiv restul factorilor din paranteze.

Rata R_i ce asigură recuperarea cheltuielilor efectuate cu dispozitivul d_i, pentru timpul T, este

$$R_i = Cd_i/T.$$

Dacă în particular funcționarea componentei d_i în toată perioada T are loc fără nici o pană, adică $T < f_i < v_i$, obținem $R_i = p_i/T$ (se consideră numai prețul elementului d_i).

La stabilirea costului execuției trebuie să stea următoarele principii: un program executat pe același sistem de operare trebuie să dea același cost, independent de configurația instalației; utilizatorul trebuie să fie încărcat numai pentru acele resurse pe care le utilizează efectiv; sistemul costurilor trebuie să asigure amortizarea cheltuielilor.

De asemenea, se cere ca sistemul costurilor să nu monopolizeze prea mult unitatea centrală pentru realizarea diverselor evidențe. Modelele actuale respectă unul sau altul din aceste principii. Ratele pot fi variabile, de exemplu, pentru a asigura repartizarea lucrărilor pe întreaga zi de lucru se poate mări R_i în orele de aglomerare (în cazul de față se va considera R_i invariant).

Prezentăm în continuare modul de determinare a ratelor diverselor elemente ale unei configurații electronice; cheltuielile de utilizare a unui dispozitiv se vor calcula cu formula (15.7).

Notînd cu T_u timpul de utilizare efectivă a instalației în perioada T, avem:

a) Rata unității centrale

$$R_{uc} = C_{uc}/T_{uc},$$

unde C_{uc} reprezintă costul unității centrale (plus cheltuielile de întreținere) iar T_{uc} este timpul mediu de utilizare a unității centrale în perioada T obținut prin măsurători statistice pentru programe caracteristice. Pentru fiecare program p_j se calculează e_j, care exprimă raportul utilizării unității centrale pe perioada execuției programului

$$e_j = t_{juc}/t_j, \qquad (15.8)$$

unde t_{juc} este timpul de utilizare efectivă a unității centrale în intervalul de timp t_j.

De exemplu, programul CONTABIL din sistemul SIRIS furnizează timpul de utilizare a unității centrale de la începutul execuției unei lucrări pînă la sfîrșitul ei.

Se face media valorilor e_j și se obține e_{uc}, rata medie a utilizării unității centrale

$$e_{uc} = (e_1 + e_2 + \ldots + e_n)/n,$$

unde n este numărul programelor caracteristice de mai sus. Timpul mediu de utilizare a unității centrale se calculează astfel deci:

$$T_{uc} = e_{uc} \cdot T_u,$$

deci

$$R_{uc} = C_{uc}/e_{uc} \cdot T_u.$$

Astfel, dacă un program utilizează unitatea centrală un timp t_{uc}, costul utilizării ei este $t_{uc} \cdot R_{uc}$.

b) Rata memoriei centrale

$$R_m = C_m/S_m \cdot T_u,$$

unde C_m reprezintă costul memoriei centrale (plus cheltuielile de întreținere); S_m — capacitatea memoriei centrale.

Se observă că utilizarea memoriei se caracterizează prin capacitatea ei și timpul de folosire.

Cum timpul de ocupare a memoriei de către un program depinde și de prezența altor utilizatori (într-un context de multiprogramare) trebuie găsit un timp echivalent, t_m, care aproximează timpul real în care programul ocupă zona de memorie, dacă ar fi singur în memoria centrală. Timpul t_m se poate calcula astfel

$$t_m = t_a + t_{uc} + t_{ii},$$

unde t_a reprezintă timpul necesar pentru activarea programului (scheduling) considerat constant; t_{uc} — timpul de utilizare a unității centrale (dat de programul CONTABIL, în sistemul SIRIS); t_{ii} — timpul în care s-au efectuat operații de intrări/ieșiri și care poate fi aproximat astfel

$$t_i = \sum_j N_j \cdot \lambda_j,$$

unde N_j reprezintă numărul operațiilor de intrări/ieșiri efectuate pe dispozitivul d_j iar λ_j este timpul mediu pentru a efectua o operație de intrare/ieșire pe dispozitivul d_j.

Cheltuielile privind utilizarea unui spațiu s de memorie într-o perioadă de timp t sînt date de produsul $s \cdot t \cdot R_m$.

În cazul sistemelor de operare în multiprogramare cu partiții fixe, s este chiar capacitatea partiției în care se execută programul.

c) Rata unității de discuri

$$R_d = (C_d + C_{ud})/N_d$$

unde C_d reprezintă costul discului (plus cheltuielile de întreținere), C_{ud} — costul unității de control a discului (plus cheltuielile de întreținere), N_d — numărul mediu de intrări/ieșiri executate pe disc în toată perioada T_u.

Ca și în cazul unității centrale, N_d se poate aproxima prin măsurători statistice pentru programe caracteristice celor ce se execută în cadrul configurației. Se calculează f_j, care exprimă raportul utilizării unității de discuri în timpul execuției programului p_j

$$f_j = N_{jd}/t_j,$$

unde N_{jd} este numărul de intrări/ieșiri efectuate în timpul t_j de execuție al programului. Se face media valorilor f_j și se obține e_d asemănător formulei (15.2)

$$R_d = (C_d + C_{ud})/e_d \cdot T_u.$$

Utilizarea discului va trebui să fie taxată prin calcularea numărului de intrări/ieșiri n_d efectuate în cursul execuției programului. Costul este dat de produsul $n_d \cdot R_d$.

d) Analog unității de discuri se pot stabili ratele altor periferice: bandă (R_b), cititor de cartele (R_c), imprimantă (R_{imp}) etc.

e) Rata altor cheltuieli

$$R_i = (C_s + C_r)/T_u,$$

unde C_s reprezintă suma altor cheltuieli (în care cheltuielile cu întreținerea și funcționarea configurației dețin o pondere principală), C_r — retribuția personalului.

Rata R_i constituie cota de repartizare a cheltuielilor comune ale secției exploatare.

Integrarea acestor rate într-un tot unitar permite obținerea formulei de calculare a costurilor prelucrării:

$$C = t_1 \cdot R_{uc} + s \cdot t_2 \cdot R_m + \sum_{i=1}^{n} N_j \cdot R_j + t_3 \cdot R_j, \qquad (15.9)$$

unde t_1 reprezintă timpul de utilizare efectivă a unității centrale pe durata totală a lucrării, s — capacitatea memoriei ocupată de program; t_2 — timpul echivalent de ocupare a memoriei centrale pe durata totală a lucrării; n — numărul de periferice utilizate; N_j — numărul de intrări/ieșiri pe perifericul j; t_3 — durata totală a lucrării.

Adaptarea modelului se face ținînd seama de posibilitatea separării cheltuielilor directe de cele indirecte, precum și de cerințele imputării raționale ale acestora. De exemplu, în anumite sisteme de operare, luarea în considerare a ratei memoriei obligă la un calcul minuțios și uneori costisitor. În aceste condiții cheltuielile cu memoria se vor include în grupa altor cheltuieli, renunțîndu-se la o rată separată.

Pentru aplicarea în practică a modelului prezentat se cere: adaptarea lui în condițiile concrete ale fiecărui centru de calcul (particularitățile sistemului de operare); extinderea sistemului de programe standard pentru a permite furnizarea tuturor informațiilor privind utilizarea resurselor în procesul prelucrării; determinarea unor rate cît mai exacte și actualizate.

De asemenea, este necesar ca fiecare fază a ciclului de viață a unui produs-program să fie prezentată distinct și precizată cu această ocazie metodologia de lucru.

Se vor menționa, de asemenea, verificările tehnice și de conducere ce se fac pe parcursul elaborării produsului-program.

Trebuie să se insiste asupra obiectivelor urmărite în întregul proces de programare și anume: minimizarea costului de fabricație; minimizarea costului de întreținere; minimizarea costului de dezvoltare.

Dimensiunea echipei de programare constituie o problemă rămasă încă nerezolvată. Aceasta și pentru faptul că nu putem vorbi de un sistem de indicatori care să permită o evaluare științifică a necesarului de forță de muncă. De asemenea, nu s-au făcut încă formulări exprese cu privire la dimensiunea optimă a echipei de programare pornind de la structura personalului și relațiile care se stabilesc între acestea în procesul muncii.

Stilul de muncă constituie, de asemenea, un factor uman deosebit de important. Se știe că elaborarea unui produs-program este o activitate cu frecvente reveniri asupra unor faze pentru completări sau perfecționări. Se apreciază chiar că *dezvoltarea prin extensii în pași succesivi* devine un principiu în programare. Așa cum am mai menționat activitatea de programare se desfășoară în echipă ceea ce presupune comunicare permanentă. De multe ori șeful de proiect procedează la o primă repartizare a sarcinilor fără să dispună de fișa tehnică a fiecărui modul și fără să definească în totalitate cerințele privind comenzile pe care acesta trebuie să le execute.

Cunoașterea problemei se realizează în aceste cazuri pe parcurs prin cercetarea documentației bibliografice și prin comunicare permanentă între programatori și între aceștia și programatorul șef de proiect.

Programatorii trebuie obișnuiți să lucreze în colectiv și să comunice permanent.

Stilul individualist, artizanal are implicații nefavorabile în planul eficienței întregului produs-program chiar dacă la nivelul unei componente a acestuia se obțin rezultate bune.

15.11. Factorii umani în programare

După cum am menționat în introducere, cheltuielile cu forța de muncă dețin o pondere însemnată în costul elaborării unui produs-program. Ele pot ajunge la peste 80%. Iată de ce se impune studierea cu multă atenție a factorilor umani care au un rol important în creșterea eficienței activității de programare.

Dintre acestea menționăm: formarea permanentă a personalului; dimensionarea echipei de programare; stilul de muncă în cadrul echipei de programare.

Formarea personalului, mai mult ca în oricare alt domeniu de activitate, trebuie gîndită ca o activitate permanentă. Programatorii trebuie să fie înarmați cu cele mai noi cuceriri ale științei în domeniul programării. Aceasta și pentru că, mutațiile sînt foarte rapide iar programarea este pe cale de a deveni în scurt timp o activitate cu o bază științifică solidă și cu un suport practic considerabil.

Experiența autorilor dovedește că principalul factor de creștere a productivității muncii o constituie formarea personalului. Considerăm că și din acest punct de vedere se impun unele măsuri capabile să asigure pregătirea forței de muncă la nivelul intelectual-creativ impus de specificul acestui domeniu de activitate.

Din experiența autorilor a rezultat că este indicat ca, ori de cîte ori începe elaborarea unui nou produs-program de dimensiuni mai mari cu grafic de elaborare pe 1÷2 ani, să aibă loc ședințe de reinstruire și informare în plan metodologic pe baza literaturii de specialitate și a unor proiecte similare elaborate.

În [124] se pune pe drept cuvînt întrebarea dacă nu ar fi bine ca documentația unui proiect să cuprindă și elemente deduse din experiență care să fie transformate în învățăminte pentru cei care urmează. Principiul de a nu redescoperi de fiecare dată ceea ce alții au aflat înaintea ta este general valabil (și cu atît mai mult în domeniul informaticii).

În programul de formare a personalului este bine să se insiste asupra faptului că nu există anume un proces tehnologic care să fie folosit pentru a dezvolta toate tipurile de produse-program.

15.12. Ergonomia programelor

Un program este construit pentru a asista omul în rezolvarea unei anume probleme sau unei anume clase de probleme corespunzător domeniului de aplicație.

Succesul folosirii lui în practică de un număr tot mai mare de utilizatori ține de *ușurința* cu care utilizatorii „colaborează" cu calculatorul în procesul muncii.

Sînt numeroase exemple de programe recunoscute ca fiind de calitate și de un real ajutor omului, dar care sînt foarte puțin utilizate. Și aceasta pentru că ele sînt greu accesibile; pentru că interfața om-calculator presupune un mare volum de cunoștințe și îndeosebi o formare specială a omului și de lungă durată.

Un program este ergonomic atunci cînd răspunde în condiții de calitate și eficiență cerințelor utilizatorului și ceea ce devine important este foarte ușor de utilizat.

Se poate spune că programele, de aplicație sau de ajutor în programare, trebuie de la început proiectate și realizate astfel încît să fie ușor de utilizat. Aceasta devine cu atît mai importantă cu cît utilizarea programelor în sistemele moderne se face în regim interactiv.

Spunem că un program este ergonomic dacă: este bine documentat, ceea ce îl face și ușor adaptabil și extensibil; este corect modularizat; este însoțit de o bază de date capabilă să asigure întreținerea în condiții de cost minime; furnizează utilizatorului suficiente informații care să-i permită folosirea cu un minim de cunoștințe (la extrema acestor cunoștințe se pot limita la activarea terminalului și aducerea programului în stare activă).

În dialogul om-calculator, omului i se poate afișa la terminal mai întîi funcțiile asigurate de program și modalitatea de alegere a unei funcții. Ulterior, după alegerea unei anumite funcții i se poate preciza modul de desfășurare a dialogului pe subfuncții (dacă este cazul) sau pentru realizarea funcției.

Utilizatorul este instruit totodată de către program cum să-l utilizeze și ce să facă în cazul în care apar anumite erori sau anomalii.

Programele ergonomice conțin fie module distincte fie elemente care asigură instruirea automată a utilizatorilor.

Tehnica „meniurilor" și a „helpurilor" asigură creșterea calităților ergonomice a programelor. *La baza dialogului om-calculator trebuie să stea prin-*

cipiul consistenței. De asemenea, trebuie să se aibă în vedere minimizarea posibilităților de apariție a erorilor și greșelilor umane. În cazul în care apar erori programul trebuie să dea posibilitatea sesizării și rezolvării lor imediate.

Deși complexitatea unei interfețe poate fi proporțională cu complexitatea datelor ce intervin în ea, sistemele complexe nu au nevoie de interfețe complexe.

Sînt două motive care cauzează complexitatea unei interfețe: atracția de a face din simplu — complex; distribuirea conceperii și implementării interfeței între mai mulți programatori.

O măsură de îmbunătățire a interfaței om-mașină este ca lucrul unei persoane să fie controlat de o altă persoană neutră.

Totodată, nu trebuie confundată documentația cu interfața om-calculator.

O atenție deosebită trebuie acordată și videoformatelor care trebuie reduse substanțial la strictul necesar, fără cuvinte de politețe.

15.13. Dreptul de proprietate în programare și măsuri organizatorice pe linia comercializării programelor

Așa cum s-a putut constata încă din capitolul 1, al volumului 1, în economie se cheltuie foarte mult cu elaborarea de programe-calculator. Un program odată elaborat poate fi multiplicat cu cheltuieli minime și livrat la un număr cît mai mare de beneficiari.

În practică se întîmplă ca de cele mai multe ori primul beneficiar, cel care a lansat comanda de elaborare a unui program-calculator, să suporte în totalitate costul de elaborare, ceilalți urmînd să le suporte eventual, pe cele de adaptare și obținere a imaginii pe suport tehnic (program + documentație + + suport tehnic). Includerea programelor într-un fond național de programe, pe domenii de aplicație, și găsirea cadrului organizatoric de comercializare constituie după părerea autorilor o soluție economică capabilă să ducă la reducerea considerabilă a cheltuielilor cu integrarea informaticii în toate domeniile de activitate*).

Faptul că software-ul care se compune din programe (sursă, obiect), procedurile de operare și întreținere și documentație, pentru care s-au înregistrat însemnate *costuri* și în care s-a investit *inteligență* și *efort uman* a făcut ca să se pună și problema protecției lui prin lege. S-au făcut asemănări între liniile sursă de program (plus documentația corespunzătoare) și editarea de cărți, subliniindu-se faptul că acestea din urmă sînt protejate prin lege.

Protejarea prin lege a software-ului constituie după părerea noastră o problemă mai complexă și nici nu ne propunem să oferim în cuprinsul acestei lucrări soluții.

Cert este că lipsa protecției software-ului avantajează masa largă de utilizatori și prin aceasta extinderea folosirii unor programe standard. În continuare vom face unele referiri de ordin organizatoric capabile să creeze un cadru avantajos de producere și utilizare de produse-software.

---

*) În practică se întîmplă uneori ca mai multe întreprinderi, de multe ori din aceeași ramură de activitate, să lanseze comenzi distincte sau să dezvolte cu propriile forțe produse-program de aplicație independent unele de altele. Toate fac aceleași cheltuieli, ba unele chiar mai multe pentru că nu dispun de specialiști de înaltă calificare.

În ceea ce privește sistemele de programe complexe proiectantul — care de drept are dreptul de autor — își poate recupera cheltuielile prin încasările pentru acordarea service-ului pentru implementare, întreținere și dezvoltare pe parcurs.

În ceea ce privește software-ul de aplicație de mai mică complexitate se poate opta pentru elaborarea unor produse software standard incluse apoi în bibliotecile de programe departamentale constituite de unitățile de comercializare a acestora.

Aceste unități pot finanța astfel de proiecte și recupera cheltuielile din livrarea lor la mai mulți beneficiari, unității producătoare revenindu-i în continuare sarcina acordării asistenței de specialitate. Unitatea care comercializează produsele software de aplicație se ocupă de ținerea la zi a beneficiarilor cu toate extensiile și perfecționările efectuate.

Între unitățile de comercializare a software-ului, la nivel departamental (pe ramuri ale economiei) se pot stabili relații în vederea preluării unor produse-program existente deja.

Pe plan național pot exista unități de comercializare a produselor software de uz general, multe din ele fiind cunoscute sub numele de programe utilitare cu rol deosebit în eficientizarea activității de dezvoltare a programelor pe scară industrială.

Unitățile de comercializare software se pot organiza sub forma unor unități distincte sau a unor sectoare de activitate în cadrul centrelor de calcul coordonatoare la nivel ramură (minister). Ele pot avea pe lîngă sarcinile de comercializare a produselor software și pe cele de marketing (studiul pieții) pentru a cunoaște în permanență cerințele pieței în domeniu. Prin centralizarea și analiza acestor cerințe software el poate lansa comenzi către unități de cercetare și inginerie software sau către compartimente de cercetare și inginerie software din cadrul centrelor de calcul aparținînd aceleași ramuri industriale. Se înțelege că fiecare centru de calcul, prin propriul compartiment de cercetare și inginerie software va dezvolta în continuare programe standard, sau „pe măsura", dar va fi în același timp interesat să asigure producerea acestora cu costuri minime, lucru posibil în general prin creșterea numărului de utilizatori. Prin urmare, atît centrele de calcul cît și unitățile de comercializare a produselor software vor fi interesate într-o colaborare avantajoasă pentru ambele părți și îndeosebi favorabilă reducerii considerabile a costului produselor software.

Desigur că ființarea unei asemenea structuri organizatorice de prelucrare și comercializare a produselor software nu este posibilă decît în condițiile unor mutații în programare care să creeze cadrul tehnologic, a structurilor metodologice și de conducere pentru elaborarea pe principii industriale a programelor.

Pentru cei interesați și îndeosebi pentru cei implicați în exportul de produse software prezentăm formele de protecție legală care pot fi aplicate produselor de tip software.

Prima formă brevetul presupune că produsul software a fost brevetat. După brevetare produsul software devine public iar utilizarea lui se face fără restricții. O asemenea formă necesită timp în vederea acordării brevetului (în condițiile în care de la depunerea formelor de brevetare pînă la acordarea brevetului nu există nici o protecție a produsului software). În plus nu există siguranța acordării brevetului, ca să nu mai luăm în calcul cercetările de efectuat, în acest scop în condițiile în care „piața" este invadată cu produse software

(se apreciază că în S.U.A. un număr de peste 70% din cererile de brevetare sînt respinse). Brevetul rămîne însă forma de protecție cea mai puternică.

A doua formă constă în dreptul de reproducere copyright și constă în dreptul unui autor de a controla răspîndirea lucrării proprii, prin copiere, către alte persoane. Această formă este mult mai simplă, dar și oferă o protecție mai redusă decît brevetul. Se recomandă ca atît programele cît și secvențele de instrucțiuni subordonate să poarte specificația de proprietate (considerînd că orice modificare a liniilor sursă este în detrimentul calității programului). În țări în care se aplică această formă de protecție, ea s-a limitat doar la forma sursă, rămînînd în suspensie celelalte forme pe care le ia programul pînă la forma de execuție (caz în care programul nu mai este un text ce poate fi citit).

Numai un program scris într-un limbaj de înalt nivel sau într-un limbaj de asamblare este recunoscut ca text și poate deci face obiectul dreptului de reproducere.

Au rămas încă neprotejate, pe lîngă programele obiect, logica programului, ideile formulate, procesele, algoritmii și operațiile pe care software-ul este destinat să le efectueze.

Problema rămasă nesoluționată în această formă de protejare este cea a ansamblului om-calculator.

O a treia formă de protejare o constituie secretul de firmă și prevederile contractuale.

Această formă este recunoscută doar de un grup restrîns de persoane și este utilizată pentru a obține avantaje materiale, protecția există atîta timp cît secretul nu este dezvăluit.

Trebuie precizat totodată că prevederile contractuale între doi parteneri nu pot acționa și asupra terților.

În țara noastră secretul de firmă este realizat prin legislația privind păstrarea secretului în unitățile economice.

Așa cum am menționat, problema protecției produselor software trebuie astfel gîndită încît să constituie un factor pe linia eficientizării activității de programare și extinderii utilizării în practică a produselor-program.

Pentru „algoritmi" și „programe" corespunzătoare se cere deci o circulație cît mai liberă, asemănătoare *ideilor* și *conceptelor*, ele constituind principalele mijloace de progres și dezvoltare pe linia științifică.

Să menționăm în încheiere că în țara noastră produsele software nu sînt brevetabile.

16. Programarea proceselor concurente

16.1. Introducere

Conceptul de programare concurentă s-a lansat odată cu apariția calculatoarelor de mare capacitate, cu multe dispozitive periferice, la care s-a urmărit exploatarea în paralel a periferiei și a unității centrale. Posibilitatea de lucru în mod asincron a perifericelor, prin intermediul întreruperilor, permite programarea „concurentă" a acestora, adică partajarea procesorului între mai multe programe și efectuarea mai multor operații simultan: calcule și intrări-ieșiri. Mai tîrziu, conceptul de concurență devine de bază în realizarea și optimizarea sistemelor de operare. Concurența apare la nivel de modul de program sau program; acestea comunică între ele, își partajează date comune sau se exclud reciproc pentru acapararea unei resurse.

Vom numi *„proces"* un modul de program sau un program ce conține o structură de date și o secvență de instrucțiuni. Dacă procesele folosesc în comun resurse logice sau fizice sau comunică date între ele atunci se pune problema concurenței, a cărei rezolvare a constituit preocuparea multor cercetători în perioada anilor '60.

Inițial programul concurent a fost împărțit în procese secvențiale ce se executau asincron, iar aceasta a constituit un pas important pe calea rezolvării problemei de concurență. Mai tîrziu Dijkstra, în sistemul THE [11], a arătat că toate comunicările între module se pot realiza prin intermediul datelor comune și pentru aceasta este absolut necesar ca operațiile pe aceste variabile comune să se realizeze cîte una o dată, adică dacă un proces lucrează cu o variabilă comună, celelalte procese, care vor să lucreze cu aceeași variabilă trebuie să aștepte pînă ce se termină operația curentă pe variabila respectivă. Pentru aceasta Dijkstra a introdus noțiunea de *„regiune critică"* pentru acele variabile pe care trebuie executate operații serial.

Regiunile critice asigură excluderea mutuală a proceselor pentru accesul la resurse comune, dar nu rezolvă problema transmiterii de date de la un proces la altul. Problema comunicării între procese este destul de dificilă pentru că procesele se execută cu viteze relative necunoscute, deci un proces nu poate ști cînd un alt proces îi comunică un mesaj. Pentru rezolvarea acestei probleme Dijkstra a introdus noțiunea de *„semafor"*. Cu acest element de sincronizare nu mai are importanță ordinea în care se execută procesele: dacă un proces are nevoie de un mesaj care nu a sosit încă atunci se pune în așteptare pe semafor și va fi activat în momentul în care a sosit mesajul; dacă mesajul sosește înainte de a fi așteptat va fi stocat în variabila semaforului și se livrează atunci cînd procesul îl cere.

În anii '71—'72 s-a propus asocierea variabilei comune cu regiunea critică ce o utilizează. De exemplu, variabila CØM poate fi folosită de mai

multe procese în citire sau scriere, astfel:

```
var CØM shared
read(X): region CØM do x: = CØM + 10 end
write: region CØM do CØM + CØM + 2 end
```

Avantajul acestei descrieri este evident: compilatorul poate verifica dacă o variabilă partajată este utilizată numai în regiunile critice, astfel încît se înlătură o sursă importantă de erori în procesul destul de complicat al folosirii în comun al resurselor.

Hoare [12] a introdus conceptul de *regiune critică condițională*, a cărei execuție este amînată pînă ce o variabilă booleană satisface o anumită condiție. De exemplu, operația de emisie a unui mesaj către un alt program devine o regiune critică a cărei execuție este condiționată de valoarea variabilei booleene ce indică dacă zona tampon de recepție este disponibilă sau nu. La fel, recepția mesajului este condiționată de variabila ce arată că zona tampon conține mesajul.

Mai tîrziu s-a sugerat că interacțiunile dintre procese pot fi mai ușor manipulate dacă se grupează toate operațiile pe o structură de date partajate într-un singur modul al programului, în loc de a le expanda în tot cuprinsul programului. Astfel în 1973 a fost propus conceptul de *„monitor"* care cuprinde la un loc declarația de variabilă partajată și procedurile ce utilizează variabilele partajate. De exemplu, o zonă temporar partajată împreună cu operațiile ce se execută asupra ei (recepție/emisie mesaj) se descrie astfel în cadrul conceptului de monitor:

```
monitor buffer;
var buff: array, flag: boolean;
procedure send (mesaj);
    if not flag do;
        buff: = mesaj;
        flag:= true;
    end;
end;
procedure receive (mes);
    if flag do;
        mes: = buff;
        flag: = false;
    end;
end;
flag: = false
end;
```

Se observă că pe lîngă procedurile de utilizare a variabilelor partajate există în interiorul monitorului și instrucțiuni de inițializare a acestora (vezi ultima instrucțiuni flag: = false). Compilatorul poate verifica dacă asupra variabilei „buff" se execută doar operațiile „send" și „receive", și pe lîngă claritate, acesta este avantajul important al monitoarelor.

Dacă conceptele programării concurente au apărute mai întîi la nivelul sistemelor de operare actual utilizatorul își dezvoltă aplicații de complexitatea acestora, astfel încît programarea concurentă a devenit un procedeu uzual de lucru. Primul limbaj de programare bazat pe noțiunile de proces și monitoare a fost limbajul PASCAL CONCURENT implementat de Brinch Hansen în 1974 [13]. Un alt limbaj bazat pe aceleași noțiuni este MODULA,

elaborat de N. Wirth [14]. Limbajul MESA [18], în care modulul devine identic cu declarația de monitor, a fost folosit pentru scrierea unui sistem de operare. Limbajele PORTAL [47] și ADA [48] includ, de asemenea, elemente de programare concurentă. Cu ajutorul acestor limbaje programele concurente devin mai fiabile, mai simple și mai clare.

Cu toate acestea se poate spune că problema formalizării și rezolvării concurenței în programare este departe de a fi rezolvată, ea rămînînd mai departe ca un obiect de studiu deosebit de important pentru viitor.

Principalele probleme ale programării concurente sînt: excluderea reciprocă; sincronizarea între procese; comunicarea între procese.

În paragrafele următoare vom relua noțiunile precedente și vom arăta cum se aplică acestea pentru rezolvarea acestor probleme.

16.2. Excluderea reciprocă

Vom numi *resursă* orice entitate necesară la execuția unui proces: memorie, unitate centrală, dispozitiv periferic, fișier, subprogram, variabilă comună etc.

O resursă este *locală* dacă este utilizată de un singur proces. O resursă utilizată de cel puțin două procese se numește *comună* sau *partajabilă*. O resursă comună se numește partajabilă cu n puncte de acces ($n \geqslant 1$), dacă poate fi atribuită simultan la n procese. Dacă resursa este partajabilă cu un singur punct de acces se va numi *resursă critică*.

Dacă mai multe procese utilizează numai resurse locale se spune că sînt independente. Dacă procesele utilizează simultan o resursă (cu mai multe puncte de intrare) se spune că sînt *paralele* pentru acea resursă, iar dacă utilizează o resursă critică se spune că sînt în *excludere reciprocă* (sau mutuală). Acea secvență de instrucțiuni a procesului în care se utilizează resursa critică poartă numele de *„secțiune critică"*.

Problema excluderii reciproce este de a serializa secțiunile critice ale proceselor, adică atunci cînd un proces intră în secțiunea sa critică să nu fie întrerupt de alt proces care să utilizeze aceeași resursă critică, fără ca procesul precedent să fi renunțat la utilizarea ei. Un exemplu evident este utilizarea unei variabile comune:

```
        Proces 1                    Proces 2
-------------------------          ------
        citește x                   citește x
        y = z * x                   x: = — x
        dacă x > 0 go to B          ------
A: tipărește (—y + x)
-------------------------
B: tipărește (y — x)
-------------------------
```

Procesele 1 și 2 partajează variabila comună x. Dacă procesul 1 este activ și citește variabila x dar este imediat întrerupt de către procesul 2 după instrucțiunea de atribuire a lui y se observă că secvența de instrucțiuni ce va fi executată de procesul 1 la reluarea lui este cu totul alta, decît cea citită anterior (procesul 2 schimbă semnul variabilei comune).

Vom presupune că vitezele relative ale proceselor sînt oarecare şi că fiecare proces iese din secţiunea sa critică după un interval finit de timp. Schema de bază este următoarea:

Proces i	*Proces j*
intrare în secţiune critică	intrare în secţiune critică
secţiune critică	secţiune critică
ieşire din secţiune critică	ieşire din secţiune critică

Există mai multe metode de rezolvare a problemei de excludere mutuală; le vom prezenta în ordinea apariţiei lor.

16.2.1. Aşteptarea activă cu instrucţiunea TAS

Fiecărei resurse critice i se asociază o variabilă booleană p care ia valoarea 1 dacă resursa este ocupată sau 0 dacă resursa este liberă. Pentru testarea ei se execută o instrucţiune specială neîntreruptibilă —TAS (test and set). Secvenţa de instrucţiuni pentru accesul la o resursă critică arată astfel:

```
A: TAS p
   go to A
   --------
   acces la resursă
   --------
```

Instrucţiunea TAS testează variabila p: dacă este 1 (resursă ocupată), nu face nimic, iar programul se continuă cu execuţia instrucţiunii următoare, GO TO, care reia acelaşi proces atît timp cît resursa este ocupată. În momentul în care p devine 0, instrucţiunea TAS face $p = 1$ şi incrementează contorul de instrucţiuni pentru a sări peste instrucţiunea GO TO, la instrucţiunile de acces la resursă. Instrucţiunea TAS nu este întreruptibilă, în sensul că o dată începută execuţia ei cu citirea variabilei p nu poate fi întreruptă de alt proces, astfel încît poziţionarea lui p va fi întotdeauna corectă. Această instrucţiune există pe unele calculatoare. Se observă că prin această metodă programul rămîne activ pînă la eliberarea resursei critice, ocupînd în mod inutil unitatea centrală. Ea se aplică dacă secţiunile critice ce utilizează resursele critice sînt de scurtă durată sau nu intervin frecvent.

16.2.2. Aşteptarea activă pe o variabilă comună

Se defineşte o variabilă comună care prin valoarea ei asigură intrarea sau nu a proceselor în secţiunile critice. Fie cazul a două procese de mai jos:

```
    integer t;
    t = 1
----------
proces 1:
    A1: if t = 2 then go to A1;
        --------
        secţiune critică
        --------
        t = 2;
        --------
```

```
proces 2:
    A2: if t = 1 then go to A2;
        ---------
        secţiune critică
        ---------
        t = 1;
        ---------
```

Excluderea mutuală este garantată de indivizibilitatea operaţiei de acces la variabila t. Dezavantajul acestei soluţii este următorul: dacă procesul 1 se opreşte după instrucţiunea $t = 2$, atunci procesul 2 nu mai poate intra decît cel mult o dată în secţiunea sa critică. Execuţia unui proces este dependentă de execuţia celuilalt.

Exemplul precedent se poate generaliza la n procese [15]:

```
    integer t;
    boolean array B (o: n), C (o: n);
    for t = 0 to n do;
        B(t) = false;
        C(t) = false;
    end; i = 0;
-----------------------------
proces-i;
    Ai: B(i) = true;
    Li: if t ≠ i then do;
            C(i) = false;
            if not B(t) then t = i;
            go to Li;
    end;
    C(i) = true;
    for j = 1 to n do;
        if j ≠ i and c(j) then go to Li;
    end;
    ---------
    secţiune critică i;
    ---------
    t = 0; C(i) = false; B(i) = false;
end proces-i;
-----------------------------
```

Ca şi în cazul precedent această metodă, menţinînd ocupată unitatea centrală, nu este utilă decît dacă aşteptările proceselor sînt de durată foarte scurtă.

16.2.3. Blocarea pe variabilă comună

Unei resurse critice i se asociază o variabilă p care ia valoarea: 0 dacă resursa este liberă; 1 dacă resursa este ocupată.

Variabila p este prevăzută cu o coadă de aşteptare $f(p)$. Dacă procesul nu poate intra în secţiune critică se pune în coada de aşteptare $f(p)$. Atunci cînd un proces iese din secţiunea critică activează un proces din coada de

aşteptare (dacă există). Pentru a realiza aceste operaţii se introduc două primitive neintreruptibile, şi anume:

```
blocare (p):
    dacă p = 0
        atunci p = 1
        altfel pune procesul în coada f(p)
    sfîrşit

deblocare (p):
    dacă f(p) nu este vidă
        atunci scoate şi activează un proces din coadă
        altfel p = 0.
    sfîrşit
```

Indivizibilitatea primitivelor de blocare şi deblocare se realizează prin procedee hardware accesibile programelor: de exemplu, se ridică prioritatea unităţii centrale la o valoare corespunzătoare astfel încît să nu mai poată fi întreruptă de alte evenimente care ar face ca procesul curent să piardă controlul.

Fiecare proces trebuie să execute următoarea secvenţă de instrucţiuni:

```
--------
blocare (p)
secţiunea critică
deblocare (p)
--------
```

16.2.4. Semafoare

Metoda excluderii reciproce prin semafoare reprezintă o generalizare a soluţiei precedente. În cazul de mai sus variabila p nu ia decît două valori, zero sau 1.

Definim un semafor s ca un cuplu $(e, f(s))$, unde: e este o variabilă întreagă; $f(s)$ este o coadă de aşteptare.

Variabila e poate să ia valori pozitive, negative sau să fie nulă. Iniţial, la declararea semaforului s, se iniţializează variabila e cu o valoare mai mare ca zero.

Asupra semaforului s se execută două operaţii indivizibile: $P(s)$ pentru cererea de acces la resursa critică şi $V(s)$ pentru eliberarea unei resurse critice.

```
P(s):
    e: = e - 1;
    dacă e < 0 atunci
                blochează procesul curent;
                pune procesul în coada f(s).
    sfîrşit

V(s):
    e: = e + 1;
    dacă e ≤ 0 atunci
                scoate un proces din coada f(s);
                activează acest proces.
    sfîrşit
```

Operațiile de introducere și scoatere a proceselor din coada de așteptare, precum și blocarea și deblocarea proceselor se face de către sistemul de operare. Tipul de gestiune a cozilor depinde de sistem (FIFØ, LIFØ, priorități etc.).

Pentru a avea acces la o resursă critică, utilizînd semaforul *s* asociat ei, orice proces trebuie să execute următoarea secvență de instrucțiuni:

```
--------
P(s)
secțiune critică
V(s)
--------
```

Ca exemplu de utilizare a semafoarelor în contextul excluderii mutuale prezentăm cunoscuta problemă a cititoarelor și redactoarelor [16].

Această problemă schematizează o situație întîlnită foarte des în gestiunea fișierelor. În acest model se consideră mai multe procese paralele (*n* maximum) divizate în două clase: „cititoare" și „redactoare". Aceste procese își partajează o resursă unică, un fișier. Fișierul poate fi citit simultan de mai multe procese cititoare, dar nu poate fi modificat (scris) decît de un singur proces redactor la un moment dat. Pe de altă parte, procesele cititoare nu au acces la fișier atîta timp cît acesta este acaparat de un proces redactor. Deci, un proces redactor are acces exclusiv la fișier din punct de vedere atît al proceselor redactoare, cît și cititoare.

Schema de principiu este următoarea:

Cititoare	*Redactoare*
cerere de citire	cerere de scriere
citire	scriere
sfîrșit citire.	sfîrșit scriere.

Vom programa cazul în care procesele cititoare au prioritate față de cele redactoare. Astfel, singurul caz în care un proces cititor trebuie să aștepte este acela în care fișierul este ocupat de un proces redactor. Un proces redactor nu poate acceda la fișier decît dacă nici un proces cititor nu este în așteptare sau în curs de lectură. Soluția problemei este următoarea:

```
integer L
semaphore exm1, exm2, r
L: = 0; exm1 = 1; exm2: = 1; r: = 1;
```

cititor	*redactor*
P(exm1)	P(exm2)
L: = L + 1	P(r)
if L = 1 **then** P(r)	----
V(exm1)	scriere
----------	----
citire	V(r)
----------	V(exm2)
P(exm1)	
L: = L − 1	
if L = 0 **then** V(r)	
V(exm1)	

În procedurile de mai sus *r* este un semafor de excludere mutuală pentru procesele redactoare. El servește primului proces redactor să ocupe fișierul, ce va fi eliberat de ultimul proces cititor.

Exm1 este un semafor de excludere mutuală pentru procesele cititoare. Dacă un proces cititor este blocat pe semaforul *r*, atunci toate celelalte procese cititoare vor fi blocate pe exm1.

Exm2 asigură prioritate proceselor cititoare față de cele redactoare. Fără exm2 s-ar putea să existe mai multe procese redactoare în așteptare pe *r* înainte de primul proces cititor blocat pe *r*.

16.2.5. Regiuni critice condiționale

Regiunile critice sînt secvențe de instrucțiuni care au acces exclusiv la o variabilă comună. Deci o regiune se asociază întotdeauna cu o variabilă partajabilă și invers. Pentru a asigura accesul exclusiv la o variabilă comună prin regiuni critice trebuie efectuate următoarele două operații:

— se declară variabila ca partajabilă printr-o directivă specială, ca de exemplu în limbajul PASCAL CONCURENT:

var v: **shared real**;

— se declară regiunile critice ce au acces la variabila partajabilă, ca de exemplu:

region v **do** . ... **end.**

Toate regiunile critice asociate lui v vor avea un acces exclusiv la această variabilă. De fapt se asociază variabilei v un semafor de excludere mutuală care face ca secvența dintre **do** și **end** a declarației „region" să fie mutual exclusivă în raport cu alte regiuni critice.

Regiunile critice prezintă avantajul că asocierea datelor partajate cu operațiile de acces la ele se face cu construcții sintactice ce pot fi verificate la compilarea programului. Rezultă următoarele considerații referitoare la regiunile critice asociate aceleiași variabile v:

— un proces poate intra într-o regiune critică imediat, dacă nici un proces nu se află în regiuni critice asociate aceleiași variabile;

— cînd un proces se află într-o regiune critică, alte procese care încearcă să intre în regiuni critice asociate aceleiași variabile vor fi întîrziate;

— cînd un proces iese dintr-o regiune critică, un alt proces din cele întîrziate poate intra în regiunea sa critică.

Problema excluderii mutuale privind accesul la resurse critice capătă o rezolvare foarte simplă prin procedeul de variabile comune asociate regiunilor critice. Astfel accesul în regiuni de excludere mutuală a n procese la o resursă critică se programează astfel:

```
var R: shared boolean;
-------------
proces-1:
        ----------------------------
        region R do utilizează resursa critică end
        ----------------------------
end proces-1
-------------
proces-n
        ----------------------------
        region R do utilizează resursa critică end
        ----------------------------
end proces-n
```

Conceptul de regiune critică a fost completat cu o nouă directivă în scopul de a permite sincronizarea între procese. Este vorba de primitiva de sincronizare „await"(p)", unde *p* este o expresie booleană ce reprezintă condiția de continuare a execuției procesului: dacă *p* este „adevărat", procesul își continuă execuția, dacă nu — se pune în așteptare pînă ce alte procese redau lui *p* valoarea „adevărat". Regiunile critice ce conțin primitive **„await"**, se numesc regiuni critice condiționale. Schema de principiu este:

region v **do** ... **await** (p) ... **end**

Condiția de sincronizare *p* este o expresie booleană oarecare, putînd include elemente ale lui *v*.

Orice proces care execută o regiune critică asociată variabilei *v* poate modifica condiția de sincronizare astfel încît unul sau mai multe procese întîrziate pe condiția de sincronizare să devină pretendente la ocuparea regiunilor critice asociate lui *v*. Cînd un astfel de proces primește dreptul de a intra în regiunea sa critică el continuă execuția cu instrucțiunea imediat următoare primitivei **„await"**.

O altă variantă pentru rezolvarea problemei anterioare de excludere mutuală a n procese pentru accesul la o resursă critică, folosind regiunile critice condiționale, este următoarea:

```
var R: shared boolean;
R: = true
-----------
proces-1:
          ---------
          region R do
                     await R;
                     R: = false
          end
          utilizează resursa
          region R do
                     R: = true
          end
          ---------
end proces-1
-----------
proces-n
          ---------
          region R do
                     await R;
                     R: = false
          end
          utilizează resursa
          region R do
                     R: = true
          end
          ---------
end proces-n
```

În această soluție excluderea mutuală se realizează explicit prin intermediul variabilei booleene R, și nu ca efect implicit al regiunilor critice. Secțiunea de utilizare a resursei se încadrează între două regiuni critice care reprezintă „ocuparea" respectiv „eliberarea" resursei, după modelul tehnicii semafoarelor:

P(s)
utilizează resursa
V(s)

Această versiune are același efect ca și cea anterioară cu regiuni critice necodiționale. Ea pune în evidență însă mecanismul sincronizării între procese.

Ca alt exemplu prezentăm modul în care se rezolvă cu ajutorul regiunilor critice condiționale problema concurenței proceselor „cititoare" și „redactoare", prezentată în paragraful anterior. Reamintim că procesele cititoare pot utiliza simultan fișierul, dar procesele redactoare trebuie să aibă acces exclusiv la el. În versiunea aceasta însă procesele redactoare au prioritate față de cele cititoare:

```
var v: shared record, cit, red: integer;
var w: shared boolean;
    cititor                          redactor
region v do                          region v do
    await (red = 0)                      red: = red + 1
    cit: = cit + 1                       await (cit = 0)
end                                  end
                                     region w do
-----------------                    -----------------
    Citire                               Scriere
-----------------                    -----------------
region v do cit: = cit-1 end         end
                                     region v do red: = red - 1 end
```

Variabila partajabilă v conține două componente întregi: numărul curent de procese cititoare ce utilizează resursa (cit) și numărul curent de procese redactoare în așteptare de resursă (red). Inițial ele sînt nule. Excluderea mutuală a proceselor redactoare se realizează prin variabila w. Se observă că soluția este mai elegantă decît în cazul utilizării semafoarelor.

16.2.6. Monitoare

Monitorul este o colecție de proceduri și date partajate de mai multe procese. În mod sintetic declarațiile de monitor încep cu cuvîntul rezervat „monitor", urmat de declarațiile de date și proceduri și se termină cu o secvență de instrucțiuni ce inițializează datele monitorului. Spre deosebire de regiunea critică, monitorul concentrează într-un singur loc datele partajate și codul ce le utilizează. Există două tipuri de proceduri monitor:

— externe (de tip entry), care pot fi apelate din exteriorul declarației monitorului;

— interne (locale), care sînt apelate numai din interiorul monitorului.

Iată cum arată în PASCAL CONCURENT declarația de monitor:

```
type nume-monitor = monitor (parametri formali)
  begin
    declarare date comune
    procedure entry nume-proc (parametri formali)
              begin
              ------
              end
    procedure nume-proc (parametri formali)
              begin
              ------
              end
    ----------------------------------------
    begin
        inițializare date monitor
    end
  end
```

Operația de inițializare a datelor monitorului se face în momentul în care se declară structura monitorului.

Procesele pot executa operații pe datele partajate numai prin apelarea procedurilor de tip entry. Apelul se face specificîndu-se și numele monitorului: „nume-monitor. nume-proc". Mecanismul de monitor asigură că numai un proces la un moment dat execută o procedură monitor; despre acest proces se spune că este în monitor. Dacă un proces este în monitor atunci oricare alt proces ce apelează o procedură de tip entry va fi pus în așteptare.

Un monitor este inițializat o singură dată. După inițializare variabilele comune și parametrii monitorului există pentru toată perioada de existență a monitorului. Eventualele variabile locale sau parametrii locali există numai pe durata execuției monitorului.

Procedurile monitorului pot să-și planifice acțiunile lor prin directivele „cond. wait" și „cond. signal". Cond este o variabilă booleană declarată în monitor sub forma:

cond: **condition**

Pe aceste variabile se construiesc cozile de așteptare ale monitorului.

Operația „cond. wait" plasează apelul curent în coada de așteptare a variabilei cond și eliberează monitorul astfel încît alte procese pot să aibă acces la variabilele lui.

Operația „cond. signal" restartează prelucrarea primului proces din coada de așteptare a variabilei cond.

Cozile de așteptare sînt prelucrate după metoda primul sosit-primul-servit cu excepția cazului în care operația cond. wait specifică și o prioritate, exprimată astfel:

cond. wait + prioritate

De exemplu, să considerăm unitatea de memorie ca resursă partajabilă și să vedem cum se poate face alocarea ei prin mecanismul de monitor [18]:

```
type alocator monitor
  begin
    memory: integer
      more : condition
      procedure entry alocare (dim: integer)
    begin
        if memory < dim then more. wait
        alocă un bloc de dimensiune dim
        actualizează memory
        p: = adresa blocului alocat
        return (p: pointer)
    end
    procedure entry dezaloc (p: pointer, dim: integer)
    begin
        eliberează blocul de dimensiune dim, pointat de p
        actualizează memory
        more. signal
    end
    begin
        memory: = 262144
    end
  end
```

Dacă blocul dorit nu este disponibil, rutina de alocare pune în așteptare apelantul pe condiția „more". Rutina de dezalocare produce o semnalare pe această variabilă de îndată ce un nou bloc a fost eliberat; aceasta are ca efect reluarea unui proces ce așteaptă pe condiția „more".

În concluzie, problema excluderii mutuale poate fi rezolvată prin unul din mecanismele prezentate mai sus. În legătură cu această problemă mai rămîn de studiat și rezolvat următoarele aspecte:

— dacă un proces este distrus în cursul execuției unei secțiuni critice, riscă să blocheze alte procese. Este deci necesar de a elibera artificial secțiunea critică a unui proces înainte de a-l distruge;

— un proces nu trebuie să rămînă un timp infinit în secțiunea sa critică, din același motiv ca mai sus;

— gestiunea cozilor de așteptare trebuie să se facă în așa fel încît să se asigure intrarea în secțiunea critică fiecărui proces (de exemplu FIFO).

Aceste probleme se rezolvă la nivelul sistemelor de operare.

16.3. Sincronizarea între procese

Sincronizarea presupune ordonarea în timp a execuției proceselor în scopul obținerii rezultatului dorit. Ea se referă la următoarele operații: blocarea unui proces; activarea unui proces blocat și eventual transmiterea de informații.

Se cunosc două metode de a realiza sincronizarea:

— direct: se acționează direct asupra procesului, blocîndu-l sau deblocîndu-l;

— indirect: se acționează asupra unor mecanisme ce implică blocarea sau deblocarea proceselor.

În principiu sincronizarea directă se realizează cu ajutorul a două primitive indivizibile:

blocare (p): trece procesul p în stare de așteptare pînă la execuția unei primitive de activare;

activare (p): trece procesul p din starea blocată în starea activă.

De exemplu, în sistemele AMS și MINOS pentru minicalculatoarele I-100, cele două directive sînt:

STOP (p): blocare proces p;

UNSTOP (p): deblocare proces p.

În limbajul FORTRAN, sub sistemul AMS pentru minicalculatoarele I-100, există și alte rutine de blocare și activare procese:

SUSPND: procesul curent se suspendă;

RESUME (p): se reia procesul p, care s-a suspendat;

REQUES(p): se cere activarea procesului p.

Dacă la sincronizarea directă se transmite explicit numele procesului de blocat sau activat, la sincronizarea indirectă se acționează asupra unor obiecte intermediare cunoscute numai de procesele ce se sincronizează. Aceste obiecte intermediare pot fi: indicatori de evenimente; contori de evenimente; semafoare; variabile comune.

În continuare vom prezenta mai multe procedee de sincronizare indirectă.

16.3.1. Sincronizare prin indicatori de evenimente

Evenimentele pot fi definite ca schimbări în starea proceselor: terminarea unei operații de intrare-ieșire, schimbarea valorii unei variabile, terminarea unui proces etc.

Într-un limbaj de programare evenimentul este reprezentat printr-un identificator ce poate fi manipulat de anumite operații speciale. Un eveniment poate fi așteptat sau declanșat. De aceea de cele mai multe ori evenimentul este reprezentat printr-o variabilă booleană, numită și indicator de eveniment, care ia două valori: 1 — dacă evenimentul s-a produs; 0 — dacă evenimentul nu s-a produs.

Un proces se poate bloca în așteptarea unui eveniment; declanșarea unui eveniment poate debloca unul sau mai multe procese.

În limbajul PL/1, cu obțiunea „task", se poate declara explicit un eveniment:

ev **event**,

sau implicit la lansarea în execuție a unui proces:

call prog **event** (ev)

Asignarea unei valori indicatorului de eveniment se face cu pseudovariabila de tip „completion" astfel:

completion (ev) = '0'B: evenimentul nu s-a produs;
completion (ev) = '1'B: evenimentul s-a produs.

Așteptarea unui eveniment, deci blocarea procesului, se face astfel: **wait** (ev).

Procesul se reia după ce evenimentul s-a produs. Dacă evenimentul s-a produs înainte de execuția instrucțiunii wait, atunci aceasta va fi inefectivă.

Așteptarea poate avea loc pe o reuniune de indicatori de evenimente. De exemplu, fie secvența de instrucțiuni PL/1 următoare:

```
————————————
call proc1 event (ev1);
call proc2 event (ev2);
wait (ev1, ev2) (i);
```

După lansarea proceselor proc1 și proc2, procesul curent așteaptă declanșarea următoarelor evenimente: dacă i = 2 se așteaptă ev1 și ev2; dacă i = 1 se așteaptă ev1 sau ev2; dacă i ⩽ 0 nu se așteaptă nici un eveniment.

În sistemul MIX [8] există 96 de indicatori pentru reprezentarea evenimentelor, din care numai ultimii 64 (33—96) se pot folosi pentru sincronizarea între programe (indicatori globali). Manipularea indicatorilor de evenimente și sincronizarea între programe se poate face prin următoarele rutine accesibile din limbajul FORTRAN—MIX:

SETEF (ev) — ev = 1: evenimentul s-a produs;
CLREF (ev) — ev = 0: evenimentul nu s-a produs;
WAITFR (ev) — așteptare eveniment ev;
READEF (ev) — citire indicator de eveniment.

În exemplul de mai jos se prezintă sincronizarea execuției a două programe:

Program A	*Program B*
CALL SETEF (45.)	CALL WAITFR (45.)
CALL DECLAR	

Inițial, indicatorul de eveniment 45 este la 0. Execuția programului B este condiționată de poziționarea acestui indicator de către programul A. În plus, programul A execută directiva DECLAR prin care declară un eveniment semnificativ în urma căruia sistemul acordă procesorul programului celui mai prioritar, gata de execuție. Acesta poate să fie eventual programul B.

Uneori se poate vorbi de întîrzierea execuției unui program ca de un eveniment particular. De exemplu, în limbajul PL/1 un proces se poate bloca pentru o perioadă dată de timp definită astfel:

delay (expresie),

unde expresie reprezintă numărul de milisecunde.

În limbajul FORTRAN-MIX există rutina MARK de forma

MARK (indic, intv, unit, cod)

unde indic este numărul indicatorului de eveniment; intv — mărimea intervalului de timp; unit — unitatea de măsură a intervalului de timp precedent; cod — codul de retur cu privire la execuția rutinei.

Prin execuția acestei rutine se pune indicatorul de eveniment la zero, iar după scurgerea intervalului de timp se pune la 1 și se declară un eveniment semnificativ. Programul se poate pune în așteptare pe perioada de timp specificată în MARK astfel:

WAITFR (indic).

Programul iese din așteptare după poziționarea indicativului de eveniment.

16.3.2. Sincronizarea prin contori de evenimente

Acest mecanism de sincronizare reprezintă o generalizare a indicatorilor de evenimente.

Un contor de evenimente este un obiect ce păstrează numărul de evenimente, dintr-o clasă de evenimente, ce s-au produs în sistem. Contorul de eveniment poate fi privit ca o variabilă întreagă nedescrescătoare.

Notăm cu c contorul de eveniment și definim următoarele primitive [19]:

— advance (c): înregistrează un eveniment din clasa de evenimente asociate contorului c. Prin execuția acestei operații valoarea lui c crește cu 1 (inițial este zero).

— read (c): se folosește pentru a afla valoarea contorului c. Programul are posibilitatea să aștepte, ciclînd pe această instrucțiune pînă ce contorul c atinge o anumită valoare, adică pînă ce un eveniment s-a produs de un număr de ori.

— await (c, v): se folosește pentru a bloca execuția programului pînă ce contorul de evenimente atinge valoarea *v*. Prin această operație se evită ca așteptarea procesului să fie activă. Dacă v = c operația await este inefectivă

Pentru a ilustra modul de utilizare al contorului de evenimente considerăm problema sincronizării a două procese din cunoscutul algoritm producător-consumator, în care se utilizează un buffer circular cu *N* locații pentru mesaje: procesul producător stochează mesajele sale în buffer (*i* modulo *N*), iar procesul consumator citește mesajul din buffer. Condițiile de lucru sînt următoarele: consumatorul citește întotdeauna un mesaj depus de producător; producătorul nu depune mesajul într-o locație, dacă aceasta mai conține un mesaj ce nu a fost încă consumat.

Se folosesc două contoare de evenimente pentru a sincroniza producătorul cu consumatorul.

— PRO, care este incrementat de procesul producător ori de cîte ori a depus un mesaj în buffer;

— CON, care este incrementat de procesul consumator ori de cîte ori a scos un mesaj din buffer.

Schema de sincronizare este următoarea:

```
    producător
integer i;
for i = 1 to infinit do;
    await (CON, i—N);
    depune mesaj în buffer(i mod N);
    advance (PRO);
end
```

```
    consumator
integer i;
for i = 1 to infinit do;
        await (PRO, i);
        scoate mesaj din buffer(i mod N);
        advance (CON);
end
```

Mecanismul de sincronizare prin contoare de eveniment nu asigură însă excluderea mutuală; el este util numai pentru ordonarea execuției proceselor care nu au nevoie în același timp de excludere reciprocă la accesul de resurse critice.

Uneori este nevoie de a stabili ce evenimente trebuie să se producă mai întîi în scopul de a avea o anumită secvență de execuție a proceselor. Pentru a obține acest lucru și, totodată, pentru a realiza excluderea mutuală a proceselor se introduce primitiva:

ticket (ST)

Această primitivă returnează valoarea curentă a variabilei *S*, numită variabila de serializare. Ea este o variabilă întreagă, inițial zero, care se incrementează ori de cîte ori este citită cu primitiva ticket. *S* poate fi privită ca un distribuitor de tickete, iar ordinea de execuție a proceselor este înscrisă pe ticketul obținut prin operația ticket (S). De exemplu, să reluăm algoritmul producător-consumator în care există însă mai mulți producători. Cu ajutorul „distribuitorului de tickete" se realizează atît sincronizarea producătorilor cît și excluderea mutuală în ceea ce privește accesul la buffer. Fiecare producător obține mai întîi un ticket care îi permite să depună mesajul în buffer, după care așteaptă să-și stocheze mesajele toți ceilalți producători cu număr de tichet mai mic. Fiecare producător va executa algoritmul următor [19]:

```
    producător i
integer t;
for k = 1 to infinit do;
    t: = ticket(S)                    } sincronizare cu ceilalți producători
    await (PRØ, t)                    }
    await (CON, t—N + 1)              } sincronizare cu
    depune mesajul în buffer(t + 1 mod N) } consumatorul
    advance (PRØ)
end
```

Consumatorul execută algoritmul prezentat precedent.

Așa cum se remarcă în [19], operațiile P și V pe semafoare pot fi realizate cu ajutorul primitivelor de lucru pe contoare de evenimente și a distribuitorului de tichete.

Fie c un contor de evenimente (inițial 0), S distribuitorul de tichete (inițial 0) și sem un semafor a cărui valoare inițială este *n*. În acest caz operațiile P(sem) și V(sem) se pot simula astfel:

```
P(sem):                    V(sem):
t: = ticket (S)                advance (c)
await (c, t — n + 1)
```

Operația „**ticket**" asignează numere fiecărui proces ce execută operația P. Procesul așteaptă pînă ce numărul corespunzător este anunțat de primitiva „**advance**" din procedura V. Diferența dintre valorile curente ale lui t și c corespunde valorii semaforului.

16.3.3. Sincronizarea prin semafoare

Pentru sincronizarea între procese prin semafoare se folosește același mecanism ca și în cazul excluderii mutuale (vezi 16.2.4.): evenimentul de activare se așteaptă prin operația P și se transmite prin operația V:

```
                    semaphore S;
                    S: = 0;
proces A                                proces B
————————                                ————————
P(s)                                    V(s)
————————                                ————————
```

Ca și în cazul contorilor de evenimente, sincronizarea prin semafoare permite memorarea numărului de evenimente. De exemplu, dacă procesul B de mai sus execută operația V(s) de n ori înainte ca procesul A să ajungă să execute operația P(s), atunci procesul A se va pune în așteptare de eveniment numai după ce a executat de n ori operația P(s);

Un semafor s este propriu (sau privat) unui proces dacă numai acel proces execută operația P(s) iar celelalte procese execută doar operația V(s). Exemplu de utilizare al semaforului propriu: activarea procesului A este dependentă de valoarea funcției booleene $f(a_1, a_2)$:

```
integer a;
semaphore exm, sempriv;
a: = 1; exm: = 1; sempriv: = 0;
Proces A                                Proces B
————————————————————————                ————————————————————————
P(exm)                                  P(exm)
if f(a1, a2) then V (sempriv)           modificare a1, a2
  else a: = 0                           if f(a1 a2) ∧ a then V (sempriv)
V(exm)                                  V(exm)
P(sempriv)
————————————————————————                ————————————————————————
```

Se observă că modificările asupra funcției booleene constituie o secțiune critică protejată de semaforul exm de excludere mutuală.

16.3.4. Sincronizarea prin regiuni condiționale

Mecanismul de sincronizare prin regiuni condiționale a fost deja descris în paragraful 16.2.5 cînd s-a prezentat directiva „await", care permite procesului să se pună în așteptare pînă ce se îndeplinește o anumită condiție. Activarea procesului cădea în sarcina sistemului.

Blocarea și activarea proceselor se poate realiza explicit dacă se implementează în alt mod directiva await și se introduce o nouă directivă, și anume [17]:

— await(v): semnifică intrarea procesului curent în coada de așteptare a variabilei v și părăsirea regiunii critice;

— cause(v): activează procesele din coada de așteptare a variabilei v.

De exemplu, să considerăm problema sincronizării mai multor procese dintr-o mulțime P, ce își alocă resurse dintr-o mulțime R. Soluția (prezentată

în limbajul PASCAL CONCURENT) este următoarea [17]:

```
var v: shared record
          available: set of R
          requests: set of P
          grant: array P of event v
       end
   procedure reserve (process: P, var ressource: R)
   region v do
       begin while empty (available) do
           begin enter (process, requests)
               await (grant (process))
           end
           remove (ressource, available)
       end
   procedure release (resource: R)
     var process: P
     region v do
           begin enter (ressource, available)
               if not empty (requests) then
               begin
                   remove (process, requests)
                   cause (grant (process))
               end
       end
```

16.3.5. Sincronizarea prin monitoare

Sincronizarea proceselor cu ajutorul monitoarelor se realizează prin directivele (vezi 16.2.6):

— cond.wait: pune procesul curent în coada de așteptare a variabilei cond și eliberează monitorul;

— cond.signal: scoate un proces din coada de așteptare a variabilei cond și îl activează.

În limbajul PASCAL CONCURENT aceste operații se realizează prin instrucțiuni „delay" și „continue". Cînd un proces este introdus în șirul de așteptare prin operația delay el pierde dreptul de acces exclusiv la monitor, permițîndu-se accesul unuia din procesele ce așteaptă în coada semaforului de acces.

Iată schema de principiu a algoritmului producător-consumator ce utilizează un buffer de N locații. Prin S se notează secvența de mesaje ce se depun în buffer. Se introduce o nouă directivă:

cond.**queue**

care returnează valoarea „adevărat" sau „fals" după cum există sau nu coadă de așteptare pe variabila cond.

Schema este următoarea [20]:

```
type prod-cons = monitor
    begin
        S: sequence of messages
        notfull, notempty: condition
        procedure entry append (x: message)
            begin
                if S.length = N then notfull.wait
                S = S ! x
                if notempty.queue then notempty.signal
            end
        procedure entry remove (result x: message)
            begin
                if S.length = 0 then notempty.wait
                x = S.first
                S = S.rest
                if notfull.queue then notfull.signal
            end
        begin
            S: = vide
        end
    end
```

Așa cum am mai spus monitoarele permit o definire clară a relațiilor dintre procese, relații ce pot fi verificate la compilarea programelor.

16.3.6. Sincronizarea prin expresii de cale

Expresiile de cale [21] permit o descriere statică a condițiilor de sincronizare a mai multor proceduri, unele din ele putînd fi concurente. Vom prezenta modul cum a fost implementat acest concept în limbajul PASCAL [22], Forma generală a unei expresii de cale este:

path S **end**

unde S reprezintă o secvență de instrucțiuni sau de apeluri de proceduri separate prin simboluri speciale, cu anumite semnificații, și anume: „;" reprezintă execuție secvențială. De exemplu:

path proc1 ; proc2 ; proc3 **end**

Procedurile proc1, proc2, proc3 se lansează în ordinea în care apar în expresia de cale: mai întîi proc1, apoi proc2, apoi proc3. Dacă un proces cere, de exemplu, execuția procedurii proc2 atunci el este pus în așteptare pînă ce un alt proces cere și termină execuția procedurii proc1.

— n: () limitarea numărului de apeluri pentru o procedură.

De exemplu:

path 3:(proc) **end**

Se permit numai 3 apeluri simultane ale procedurii proc, ce se pot executa concurent. Orice alt proces care invocă proc va aștepta pînă ce mai puțin de 3 apeluri sînt active. Numărul din fața parantezei poate fi interpretat ca

numărul de resurse pentru care concurează procedura. O secțiune critică, în care se partajează o singură resursă se specifică astfel:

path 1: (proc1, proc2, proc3) **end**

Numai una din cele 3 rutine pot fi active la un moment dat.

—, nu se impune nici o restricție în ordinea de execuție a proceselor. De exemplu:

path proc1, proc2, proc3 **end**

Procesele proc1, proc2, proc3 pot fi lansate în orice ordine și ori de cîte ori sînt apelate.

—[] execuție continuă. De exemplu:

path proc; [spooler] **end**

Se cere ca procesul proc să se execute înaintea procesului spooler, dar odată ce procesul spooler și-a început execuția, trebuie să continue pînă ce epuizează toate apelurile la el.

Limbajul PASCAL a fost completat, pe lîngă noțiunile de proces și expresii de cale, cu noțiunea de obiect. În cadrul definiției obiectului se specifică o expresie de cale ce asigură sincronizarea accesului la datele obiectului. Sintaxa entității obiect este următoarea:

```
<tip obiect> : : = object
                   path expresie de cale end
                       <declarare constante>
                       <declarare tipuri de date>
                       <declarare funcții, proceduri, procese>
                       <inițializare>
                   end
```

Obiectul este utilizat pentru a defini o dată abstractă, operațiile ce se pot executa pe componentele obiectului și sincronizarea între aceste operații. Compilatorul PASCAL verifică dacă fiecare operație specificată în expresia de cale are o definiție în corpul obiectului. Corpul obiectului include, de asemenea, declarații de variabile și eventual funcții și proceduri locale, inaccesibile din afara obiectului. Un obiect poate include opțional o procedură de inițializare a variabilelor sale: ea este delimitată de cuvintele rezervate **init** și **end**.

Să considerăm obiectul „buffer" și procedurile de acces la el, de tip producător-consumator. Dimensiunea bufferului este de n locații. Redăm schema din [22]:

```
buffer = object
         path n: (1:(producător); 1:(consumator)) end;
         var iobuffer: array (1 ... n) of message;
             inp, out: 1 ... n;
         entry procedure producător (m: message);
             begin
                 iobuffer (inp): = m;
                 inp = inp + 1 mod n;
             end;
```

```
entry procedure consumator (var m: message);
    begin
        m: = iobuffer (out);
        out: = out + 1 mod n;
    end
init;
    begin
        inp: = 1;
        out: = 1;
    end;
end;
```

În expresia de cale de mai sus se precizează că numai n apeluri pot exista la un moment dat și că procedura producător precede procedura consumator.

Campbell și Habermann au indicat o metodă de translatare automată a expresiilor de cale în semafoare [23]. Expresiile de cale, împreună cu definiția obiectului, constituie, la fel ca monitoarele, o metodă simplă și clară de sincronizare a proceselor, ce poate fi verificată la compilarea programelor. Pe de altă parte primitivele de sincronizare sînt generate automat, eliminîndu-se astfel erori grave de programare.

16.4. Comunicarea între procese

Metodele de sincronizare prezentate mai sus permit o cooperare între procese numai în sensul blocării și activării lor reciproce. În practică, acest mod de cooperare nu este suficient, este necesar ca procesele să schimbe informații între ele. Dar comunicarea de date este strîns legată de procesul sincronizării. De aceea problema comunicării de informații se va împleti cu problema sincronizării proceselor.

16.4.1. Comunicarea prin date comune

Comunicarea între programe prin date comune este destul de cunoscută de programatori. Astfel, în limbajul FORTRAN se pot specifica datele comune prin directiva COMMON. Datele comune nu ridică nici o problemă în cazul programelor ce se execută secvențial. Dacă programele se execută în paralel și accesul la datele comune nu este previzibil se pune problema excluderii reciproce, care se poate rezolva, așa cum s-a văzut în paragrafele precedente, prin includerea instrucțiunilor de acces la date în secțiuni critice. Aceasta nu rezolvă în totalitate problema comunicării pentru că de cele mai multe ori este necesar ca mesajul transmis să cuprindă și semnalul de activare al procesului destinatar (dacă acesta era blocat în așteptare de mesaj). Comunicarea de date reunește și sincronizarea între procese. Prezentăm, în continuare, cum se rezolvă această problemă cu ajutorul semafoarelor, în cazul a două procese (producător și consumator), ce comunică între ele printr-o zonă comună de memorie. Reamintim condițiile problemei: informația este constituită din mesaje de lungime constantă; nu se face nici o ipoteză asupra vitezelor relative ale proceselor; excludere mutuală la nivel de mesaj: consumatorul nu poate utiliza un mesaj pe care producătorul este în curs de a-l stoca; dacă producătorul este în așteptare din motiv că zona de comunicare este plină

trebuie să fie anunțat de îndată ce o locație pentru mesaj devine disponibilă (la fel pentru consumator, dacă acesta este în așteptare din cauză că zona tampon este vidă).

Presupunem că zona de memorie comună poate conține maximum n mesaje ($n > 0$).

Pentru sincronizarea accesului la zona comună se folosesc 2 semafoare:

— mes — inițial 0, care indică numărul de mesaje depuse în zona tampon și pot fi utilizate;

— loc — inițial n, care indică numărul de locații disponibile pentru mesaje în zona tampon.

Schema de principiu a celor două procese este următoarea:

```
                    semaphore mes, loc;
                    mes: = 0; loc: = n;
prod:                       cons:
    producere mesaj             P(mes)
    P(loc)                      scoate mesajul din zona tampon
    stocare mesaj în zona tampon V(loc)
    V(mes)                      utilizare mesaj
    salt la prod                salt la cons
```

Dacă se consideră mai multe procese „producător" și mai multe procese „consumator" atunci trebuie asigurată excluderea mutuală a proceselor de același tip la manipularea locațiilor din zona tampon. Pentru aceasta se introduc încă două semafoare de excludere mutuală: exmp pentru producători și exmc pentru consumatori. Schema finală, generalizată este următoarea:

```
semaphore mes; loc;
mes: = 0; loc: = n
semafoare exmp; exmc;
exmp: = 1; exmc: = 1;
integer ip, ic
ip: = 0;
ic: = 0;
```

Producător i	*Consumator i*
prodi:	**consi**:
producere mesaj	P(mes)
P(loc)	P(exmc)
P(exmp)	mesaj := buffer(ic)
buffer(ip) = mesaj	ic: = ic + 1 mod n
ip: = ip + 1 mod n	V(exmc)
V(exmp)	V(loc)
V(mes)	utilizare mesaj
salt la prodi	salt la consi

La fel se pot folosi și monitoarele, regiunile critice condiționale sau expresiile de cale pentru comunicarea de informații între procese, așa cum s-a văzut în exemplele din paragrafele precedente.

16.4.2. Comunicarea prin cutie poştală

Mecanismul de comunicare prin cutie poştală reprezintă o aplicaţie directă a modelului producător-consumator. Cutia poştală este o zonă tampon asupra căreia se pot face două operaţii: depunere mesaj (emisie); scoatere mesaj (recepţie).

Zona tampon poate fi statică sau dinamică. Cutia poştală poate fi comună pentru mai multe procese sau poate fi proprie unui proces. Mesajele conţin în general şi identificatorul procesului emiţător, iar dacă cutia poştală este comună pentru mai multe procese se adaugă şi numele procesului destinatar.

Problemele puse de cutiile poştale sînt legate de: modul de alocare: static sau dinamic; dimensiunea cozii de mesaje (problema umplerii cozii); accesul la cutia poştală; ea trebuie adresată de toate procesele care o utilizează, ceea ce este dificil în cazul în care fiecare proces dispune de un spaţiu de adresare distinct.

Alocarea statică a cutiei poştale presupune rezervarea unor zone de memorie suficient de mare în care se depun mesajele. Ea se foloseşte mai ales în cazul în care mesajele sînt de dimensiune fixă şi se poate aprecia numărul maxim de mesaje (fig. 16.1).

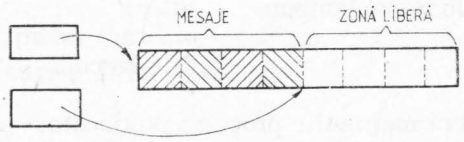


Fig. 16.1. — Reprezentarea alocării statice a cutiei poştale.

În general, alocarea spaţiului pentru cutia poştală se face în mod dinamic pe măsura sosirii mesajelor. De exemplu, în sistemul MIX fiecărui program se alocă o cutie poştală, iniţial vidă, formată din două cuvinte care conţin pointeri pe primul, respectiv ultimul mesaj (fig. 16.2).

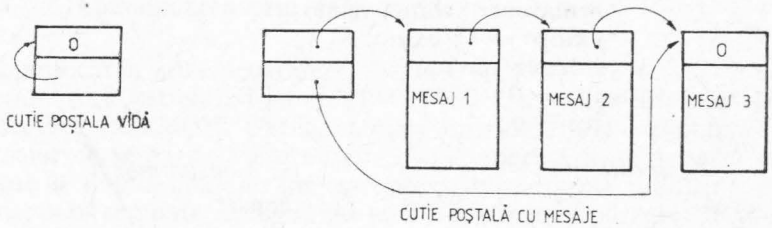


Fig. 16.2. — Reprezentarea alocării dinamice a cutiei poştale în sistemul MIX.

Cutia poştală, împreună cu mesajele ce se depun în ea, se află într-o zonă de memorie comună, numită zonă de alocare dinamică. Depunerea mesajelor în cutia poştală, precum şi scoaterea lor se face sub controlul sistemului de operare, la iniţiativa programului prin directive speciale. De exemplu, în limbajul FORTRAN-MIX se pot apela următoarele rutine:

— SEND (p, m), prin care se pune mesajul m în cutia poştală a programului p. Mesajul se depune indiferent de starea programului, activ sau în aşteptare, deoarece cutia poştală există pe toată durata instalării programului în sistem.

— RECEIV (p, buf), prin care se scoate din cutia poștală proprie mesajul emis de programul p și se depune în bufferul buf. Dacă nu se specifică programul p se scoate primul mesaj din cutia poștală.

În general, programul destinatar nu este prevenit de introducerea mesajelor în cutia sa poștală. În unele sisteme, ca de exemplu în MIX, programul destinatar poate obține imediat mesajul deoarece punerea lui în cutia poștală constituie un eveniment ce provoacă o întrerupere prin care se dă controlul unei rutine speciale din programul destinatar pentru prelucrarea mesajului [8]:

Exemplu de utilizare a cutiei poștale.

Se dorește exploatarea în paralel a mai multor unități dintr-o aplicație complexă: mai multe procese utilizator; un proces special ce citește fișiere de pe disc; un proces atașat imprimantei, ce listează fișierele.

Procesele utilizator emit comenzi (mesaje) către perechea de procese disc-imprimantă, pentru listarea unor fișiere de pe disc (fig. 17.3).

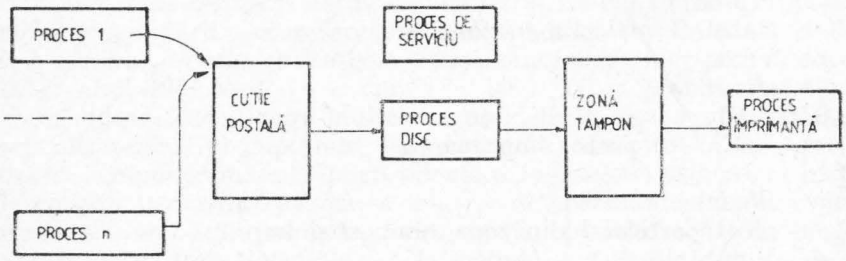


Fig. 16.3. — Utilizarea cutiei poștale.

Comenzile proceselor utilizator se depun într-o cutie poștală, din care se scot de procesul disc ce activează, la rîndul lui, procesul imprimantă. Accesul la cutia poștală este de tip producător-consumator. Manipularea cutiei poștale se face astfel:

— Înaintea oricărei depuneri de mesaj se verifică dacă cutia poștală este plină. Cutia se alocă static și are o dimensiune maximă n.

— Dacă cutia poștală este plină se activează un proces de serviciu ce imprimă un mesaj de eroare.

— După ce procesul utilizator depune mesajul în cutia poștală anunță procesul disc prin executarea operației V asupra unui semafor mes, inițial la zero.

Deoarece mai multe procese au acces la cutia poștală trebuie asigurată excluderea mutuală. Pentru aceasta se va utiliza semaforul exm, cu valoarea inițială 1.

Procedura de acces la cutia poștală, a fiecărui proces utilizator, este următoarea:

```
proces i
-----------------
P(exm);
dacă cutie — poștală—plină
      atunci eroare: = „adevărat"
      altfel depune mesajul în cutia poștală;
             eroare: = „fals";
sfîrșit V(mes)
V(exm)
-----------------
```

Cutia poștală este gestionată ca un tampon circular. Există un pointer de coadă comun pentru toate procesele utilizator și un pointer de început de mesaje pentru procesul disc.

Mecanismul cooperării între procesul disc și procesul imprimantă este, de asemenea, de tip producător-consumator. Se utilizează două semafoare: prod (inițial m) și con (inițial 0). Dimensiunea zonei tampon dintre procesul disc și imprimantă este m:

proces disc

```
pd:
    P(mes);
    scoate un mesaj din cutia poștală;
    pînă la sfîrșit de fișier execută;
    P(prod);
    transferă articolul în zona tampon;
    V(con);
    sfîrșit
salt la pd
```

proces imprimantă

```
pi:
    P(con);
    scoate articolul din zona tampon și imprimă-l;
    V(mes);
    salt la pi.
```

16.4.3. Comunicare prin semafoare cu mesaje

Principiul semaforului cu mesaj este următorul: în momentul în care se execută operația V asupra semaforului propriu unui proces se transmit și informații procesului respectiv. Se definesc două primitive P_M și V_M astfel: P_M este operația P completată cu recepția unui mesaj; V_M este operația V completată cu emisia unui mesaj.

Pentru descrierea acestor operații se introduce o nouă variabilă de tip mesaj care împreună cu variabila semaforului apare ca parametru în procedurile lui P_M și V_M.

De asemenea, se asociază semaforului două șiruri de așteptare: un șir f_p de procese și un șir f_m de mesaje.

Operațiile P_M și V_M se pot descrie astfel, notînd cu p procesul ce le execută:

```
PM(s, mesaj)
e(s): = e(s) — 1;
dacă e(s) < 0 atunci
    blochează procesul P;
    introduce p în lista fp(s);
altfel
    scoate un mesaj m din fm(s);
    mesaj: = m;
```

```
V_M(s, mesaj)
e(s): = e(s) + 1;
dacă e(s) ≤ 0 atunci
      scoate un proces q din f_p (s);
      transmite mesajul lui q;
      activează procesul q;
altfel
      introduce mesajul în f_m(s).
```

Inițial $e(s)$ este nul, iar șirurile de așteptare $f_p(s)$ și $f_m(s)$ sînt vide. Dacă se inițializează $e(s)$ cu valoarea n, atunci trebuie inițializat șirul f_m de mesaje cu n mesaje.

*

Complexitatea aplicațiilor software actuale, precum și necesitatea de a folosi mai bine resursele hardware impun utilizarea tot mai largă a mecanismelor programării concurente. Limbajele concurente ca PASCAL, MODULA, MESA au avut un impact deosebit asupra conceperii programelor concurente. Dacă în anul 1962 încă nu se cunoștea și stăpînea problema „îmbrățișării mortale" (dead lock), acum multe din problemele concurenței sînt lămurite și formalizate. Saltul important s-a realizat în momentul în care programul a fost descompus în module independente ce se execută asincron. În programele concurente modularizarea se impune din necesitatea de a stăpîni și a controla sincronizarea între diverse operații, spre deosebire de programele secvențiale, la care descompunerea în module se face din dorința de a facilita verificarea programului sau a-l face mai clar.

Trebuie subliniat faptul că validitatea utilizării mecanismelor de sincronizare este condiționată în ultimă instanță de comportarea și fiabilitatea proceselor care le folosesc. De exemplu, este posibil ca primitivele de sincronizare să fie utilizate în mod incorect de către programator și să se ajungă la situații de blocare definitivă a unor resurse critice. În acest timp, problema verificării automate a corectitudinii proceselor concurente devine foarte importantă în vederea detectării interblocajelor sau a funcționării defectuoase a unor procese paralele. Un astfel de verificator automat este prezentat în [49]. Dintre mecanismele de sincronizare prezentate în paragrafele precedente expresiile de cale conduc la generarea automată a utilizării primitivelor de sincronizare, astfel încît programatorul este eliberat de problema implementării lor ceea ce constituie un cîștig important în eliminarea erorilor de programare. În cazul semafoarelor, utilizarea primitivelor P și V este lăsată în seama utilizatorului ceea ce implică un procent ridicat de erori la nivelul de fiabilitate al soluțiilor obținute. Expresiile de cale apar ca o metodă de abordare structurată a descrierii obiectelor partajate și a sincronizării și comunicării dintre programe.

Totuși, mecanismele de comunicare și sincronizare prezentate anterior nu satisfac în totalitate cerințele mereu crescînde ale utilizatorilor. Astfel în limbajul de programare ADA apare un nou concept de sincronizare, conceptul de „rendez-vous" [48]. Sincronizarea între procese se realizează printr-un procedeu asemănător apelului de procedură. Într-un proces punctul de rendez-vous se indică prin apelul unei subrutine din alt proces, specificîndu-se numele procesului și numele procedurii. În procesul ce conține procedura apelată se marchează punctul de sincronizare prin instrucțiunea „**accept**" și

numele procedurii care cuprinde un şir de instrucţiuni cuprinse între „**do**" şi „**end**". Schema de principiu este următoarea:

```
        proces P1                          proces P2
  ----------------                   ----------------
punct de
rendezvous P2.PRØCX (parametri)   accept PRØCX (parametri)   punct de
                                        do                    sincronizare
  ----------------                   -----
                                        end
                                   ----------------
```

Cînd în cursul execuţiei, un proces ajunge la punctul de rendez-vous sau de sincronizare va aştepta pînă cînd şi celălalt proces ajunge la punctul corespunzător. După realizarea acestei sincronizări se execută o acţiune de apel de subprogram (PRØCX din P2) realizîndu-se şi o comunicare de date prin intermediul parametrilor. După terminarea execuţiei secvenţei de instrucţiuni dintre do şi end din procedura apelată cele două procese îşi reiau activitatea în paralel.

Dacă mai multe procese apelează aceeaşi procedură înainte ca procesul apelat să fi ajuns la instrucţiunea accept corespunzătoare, atunci apelurile sînt introduse într-un şir de aşteptare şi vor fi servite în ordinea strictă a sosirii, cîte unul la fiecare execuţie a instrucţiunii accept.

Tot în limbajul ADA un program poate aştepta mai multe evenimente prin intermediul instrucţiunii „select". Tratarea evenimentelor se face în mod selectiv, adică nu neapărat în ordinea apariţiei lor. Descrierea formală a instrucţiunii select este următoarea:

```
instr-accept :: =
    select
    [when condiţie =>]
        op-select
    {or [when condiţie =>]
        op-select}
    [else
        şir-instrucţiuni]
 end select;
op-select :: = instr-accept [şir-instrucţiuni] | instr-delay [şir-instrucţiuni]
```

Se observă că fiecare operaţie select conţine în mod obligatoriu o instrucţiune accept sau o instrucţiune de întîrziere a execuţiei programului o perioadă limitată de timp (delay).

O altă tendinţă ce se manifestă actual este conceperea de calculatoare specializate pentru programe concurente. Tehnica microprocesorului poate permite construirea de arhitecturi ce implementează noţiunile de proces şi monitor. De exemplu Hansen [24] propune un calculator cu 10 microprocesoare, fiecare microprocesor avînd o memorie locală dedicată unui singur proces. Microprocesoarele partajează o memorie comună ce conţine monitoarele.

În încheiere se poate spune că problema concurenţei în programare rămîne o problemă de viitor, care aşteaptă noi soluţii de rezolvare.

17. Instrumente software și medii de programare

17.1. Introducere

Așa cum s-a menționat în capitolul 1, din volumul 1, previziunile cu privire la evoluția elaborării programelor indică o creștere a rolului instrumentelor software ce pot fi folosite pe parcursul unui proiect.

Tendința este ca asemenea instrumente să vizeze deopotrivă toate etapele ciclului de viață al programelor de la întocmirea specificațiilor de definire la exploatare și actualizare. Totodată se urmărește integrarea acestor instrumente după un concept unitar într-un mediu evoluat de dezvoltare de programe.

Cele mai multe din actualele instrumente de dezvoltare software vizează cu precădere etapele de codificare și de testare a programelor. Este, așa cum vom vedea în paragraful următor, și principala limită a mediului de programare UNIX.

Literatura de specialitate [106] evidențiază ca instrumente software de bază folosite în dezvoltarea programelor *):

— *Editorul de texte* cu ajutorul căruia se creează și se actualizează interactiv texte care pot constitui date sau programe;

— *Macroprocesorul* utilizat îndeosebi pentru a diminua deficiențele unor limbaje de programare pe linia dezvoltării sistemice de programe. Spre exemplu macroprocesorul RATFOR permite structurarea programelor FORTRAN folosind structurile standard de control (secvențială, alternativă și repetitivă). Mai întîi RATFOR înlocuiește aceste macrouri cu definirea lor în termeni FORTRAN. Programul este compilat de către compilatorul FORTRAN standard.

— *Interpretor/compilator* utilizat pentru interpretarea și execuția programelor (cazul interpretorului) sau pentru analiza și trecerea programelor din formatul sursă în format obiect (cazul compilatorului).

— *Fișiere sistem* folosite pentru memorarea datelor și/sau programelor în formatul sursă sau în formatul obiect.

— *Link-editor* pentru editarea legăturilor între module de program (obținute în urma compilării sau existente într-o bibliotecă) și obținerea programului obiect în format executabil.

Alături de aceste instrumente considerate de bază apar o serie de alte instrumente software destinate să asiste programatorul și să automatizeze în proporții diferite activitățile specifice dezvoltării de programe.

În măsura în care setul de instrumente software folosite în dezvoltarea programelor posedă caracteristicile: singularitate scop; ușurință în utilizare;

---

*) În accepțiunea de codificare și testare a programelor.

niveluri de utilizare; compatibilitate cu alte instrumente; adaptabilitate; inteligență locală. ele se pot integra eficient într-un mediu de programare.

În [115] se precizează că un mediu de dezvoltare al software-ului creează posibilitatea de a dispune — în orice moment — de informații privind stadiul realizării lui; permite reutilizarea acestuia și facilitează adaptarea software-ului la noi cerințe.

Literatura de specialitate [100] clasifică mediile de programare în două mari categorii:

— *medii de programare neinteractive*, care nu dau programatorilor posibilitatea de a urmări „on-line" efectele intermediare a diverselor soluții de programare, de acces la informații cu caracter de structură, acces posibil la diferite versiuni ale aceluiași program etc.;

— *medii de programare interactive*, care asigură programatorului posibilitatea lucrului interactiv pentru a urmări efectele soluțiilor adoptate, de a avea acces la diferite versiuni ale aceluiași program sau diferite părți din program.

La rîndul lor aceste medii pot fi *universale* (mai rar întîlnite în practică) sau *orientate* pentru anumite limbaje de programare.

Referitor la mediile interactive se consideră ca elemente cheie ale acestora:

— *un nou mod de reprezentare internă a programelor*, de regulă sub forma arborilor, a tuturor versiunilor modulelor de program cu posibilitatea de a asocia fiecărui nod din arbore texte comentarii;

— *un editor orientat* pe lucru cu structuri care permite efectuarea cu ușurință a unor operații asupra arborilor de reprezentare a algoritmilor;

— *un evaluator* capabil să lucreze în diferite moduri și contexte specifice și care furnizează o serie de informații utile programatorului.

Inițial un mediu de programare (dezvoltare de programe) constă dintr-un compilator și un editor de legături. Mai tîrziu încep să apară editoare de texte, programe de depanare, metode de concepere programe, standarde de programare etc. În momentul de față numărul instrumentelor de dezvoltare de programe trece de 400 [125].

Într-un mediu de dezvoltare programare se includ instrumente corespunzătoare etapelor (fazelor) ciclului de viață al programelor.

Pentru specificarea cerințelor se precizează folosirea documentelor de specificații scrise într-o formă neformală iar de dată mai recentă a documentelor de specificații scrise într-o formă formală (vezi limbajul SADT, ADES etc.).

Instrumentele pentru concepere se diferențiază pe cele două faze: conceperea preliminară și respectiv conceperea de detaliu. Ele pot fi neformale (descriptive) sau sub formă de diagrame ca rezultat al unei metodologii formale de concepere (ca de exemplu conceperea structurată). Faza preliminară duce la generarea unor planuri de elaborare ce descriu ce module fac parte din sistem și cum vor fi construite documentele de descriere. Modulele sînt destinate persoanelor ce concep sistemul și nu utilizatorilor.

Instrumentele de codificare vizează pe lîngă codificare și operații de construire și modificare a manualelor, rapoartelor, textelor.

Instrumentele de întreținere includ specificații de configurare, proceduri de schimbare control, rezultate teste, documente de referințe încrucișate etc.

În funcție de complexitatea proiectului se apreciază că este necesar un set minim de instrumente care definesc nivelul mediului de programare.

După [121] în funcție de acest criteriu, se pot determina:

— *Mediul I* corespunzător dezvoltării de proiecte medii. Pentru specificații cerințe nu sînt necesare instrumente automate. Se pot folosi diagramele HIPO pentru definirea specificațiilor sau diagrame de ilustrare a fluxului de transformare a datelor.

Documentele de concepere trebuie să includă descrierea structurilor majore de date, fișierele și componentele funcționale ale sistemului (necesare pentru generarea datelor de test). Singurul instrument automat este dicționarul de date. Alte probleme ale procesului de concepere sînt manuale. Dicționarele de date pot fi folosite pentru înregistrarea tuturor deciziilor relative, la structura și implementarea fiecărui fișier. Sistemele de control ale codului sursă se situează în intervalul de la cele mai simple produse de introducere și regăsire a versiunilor modulelor la instrumentele ce transformă automat o versiune de program în alta. Cele mai simple programe de întreținere cod sînt editoarele de texte.

Singurul instrument automat pentru verificare sînt programele de comparare a fișierelor. Testele sînt construite manual. Pentru faza de întreținere nu se prevede nici un instrument software pentru automatizare.

— *Mediul II* corespunzător dezvoltării de proiecte mai mari. Acest mediu include și baza de date a proiectului pentru stocarea și regăsirea produselor în diferitele faze ale ciclului de viață.

La construirea bazei de date se au în vedere trei concepte: obiect; caracteristicile unui obiect; relațiile între obiecte.

Obiectele pot fi module de program, programe de test, documente de concepere, specificații cerințe sau orice alt produs software. Baza de date în sistemul PSL/PSA al proiectului ISDOS este un exemplu de o versiune specializată pentru gestiunea unui proiect.

Una dintre cele mai importante utilizări a bazei de date este modelarea și documentarea relațiilor mai importante dintre obiecte. Astfel de relații sînt relații de compilare între modulul sursă și cel obiect; relația de apel dintre un apel de procedură și procedura apelată; relația de implementare dintre o funcție și o parte de cod; relația de precedență în execuție dintre două programe; relația de cuplare cu datele dintre două module; relația de parte dintre specificație și un set de module sau fișiere.

Acest mediu conține un instrument de analiză a dependenței pentru introducerea de relații de dependență și regăsirea obiectelor ce sînt într-o relație specificate prin nume.

Relațiile de dependență pot să nu fie incluse într-o bază de date. În acest caz se examinează direct obiectele pentru a afla proprietățile sau relațiile dintre ele. Dintre instrumentele acestor medii menționăm instrumente pentru stocarea și manipularea specificațiilor (în format mașină). Definirea cerințelor se face manual. Aceste instrumente nu necesită obligatoriu utilizarea bazei de date. Dacă este utilizată, totuși se realizează introducerea automată a relațiilor de dependență.

Metodologiile folosite (HIPO, PSL/PSA, LDS/ADES etc.) presupun o formă a unei notații grafice sau un limbaj formal pentru reprezentarea specificațiilor sau o metodologie sistemică pentru generarea specificațiilor. Spre exemplu metodologia SADT* utilizează forme speciale pentru notarea componentelor sistemului, relațiile dintre componente și datele de intrare și ieșire.

---

* Structural Analysis and Design Techniques, metodă dezvoltată de D. Ross.

Pentru concepere se presupune utilizarea unor reprezentări de concepere formale iar ca metode: metoda Jackson, top-down, PDL etc.

Proiectarea structurată este corespunzătoare reprezentării componentelor funcționale și a ierarhiei de relații dintre ele.

Pentru codificare se utilizează baza de date a proiectului. Un modul sursă poate să aibă mai multe caracteristici: timpul cînd a fost introdus în baza de date; sistemul de operare necesar; calculatorul necesar; relațiile cu alte module etc.

Pentru verificare în afara comparatorului automat de fișiere mai există o serie de instrumente de verificare automată cum ar fi: analizorul de execuție a instrucțiunilor (cît timp au fost executate instrucțiunile în perioada testelor) și programe de testare pentru verificarea unor părți de cod.

În vederea întreținerii mediul de nivel II include un sistem de control automat al proiectului și asigură facilități pentru stocarea informațiilor de gestiune și generare de rapoarte.

Printre alte instrumente se citează:

— *analizorul datelor din program* depistează anomalii în program ca de exemplu utilizarea unor variabile neinițializate;

— *formatorul codului sursă* care formatează programul pentru a fi citit mai ușor;

— *analizatorul fluxului de execuție* care construiește structura arborescentă a apelului de proceduri sau subrutine;

— *verificatorul de interfețe* care verifică dacă tipul parametrului actual corespunde cu cel formal la apelul de subrutine;

— *monitorul de evaluare performanțe* care măsoară timpul de execuție al diferitelor părți dintr-un program;

— *analizorul de referințe încrucișate* care listează definițiile și referințele fiecărui obiect;

— *depanatorul simbolic* care permite depanarea programelor sursă (vizualizarea variabilelor, modificarea locații etc.);

— *programe de elaborare documentație* care realizează automat documentațiile produsului software;

— *generatoare de date de test* care analizează programul și descrie lista de date de test necesare pentru a parcurge o anumită ramură din program.

— *Mediul III* de dezvoltare programe are în general aceleași caracteristici ca și mediul II, cu precizarea că unele instrumente devin obligatorii.

— *Mediul IV* de dezvoltare programe caracterizat prin utilizarea bazei de date a proiectului. Căutarea unor obiecte cu anumite proprietăți se face cu ajutorul unui limbaj de interogare. Se dă o mare atenție modificărilor în fișiere ce implică noi versiuni. Acestea trebuie să se facă sub controlul unui program special, care cere motivul modificării și include noua versiune în baza de date.

În cadrul acestui mediu se utilizează instrumente automate de specificare cerințe, într-o manieră formală.

Definirea cerințelor și metodologiilor de specificare trebuie să includă proceduri de construcție incrementală și revizuiri.

Faza de definire cerințe și specificații poate include experimentări prin simularea unor părți de sistem.

De exemplu se poate simula interfața interactivă a unui sistem pentru ca utilizatorul să se pronunțe dacă este acceptată sau nu.

Conceperea este și ea automatizată. Reprezentările se includ în baza de date. Se pot folosi tehnici de simulare în GPSS sau SIMSCRIPT. Metodologia de concepere ca și cea de elaborare cerințe trebuie să permită construcții incrementale și proceduri formale pentru revizuire concepere. Conceperea pe faze sau incrementală este necesară în proiectele mari, deoarece diferite părți ale proiectului sînt elaborate de persoane ce nu sînt în contact permanent.

Instrumentele și tehnicile de codificare și testare sînt similare celor din mediul III de dezvoltare programe.

Pentru întreținere sînt disponibile metode pentru construire de planuri de elaborare și instrumente pentru aplicarea lor.

Multe din mediile de programare citate în literatura de specialitate fiind*) în curs de realizare sau de evaluare, în practică, se întîlnesc îndeosebi *instrumente software disparate* sau folosite într-un *sistem suport de dezvoltare software* (care în stadiul integrat definește un mediu de programare).

Spre exemplu, pentru dezvoltarea de programe FORTRAN se folosesc preprocesoare care asigură codificarea directă a structurilor standard de control. Sistemul SIRIS implementat pe calculatoarele din gama FELIX C asigură generarea automată a datelor de test. Există numeroase instrumente software de testare și depanare dinamică a programelor, de generare automată a documentației etc.

În [102] se evidențiază existența în BNP (Biblioteca Națională de Programe) a unui număr însemnat de produse-program pe următoarele categorii: limbaje de specificare și procesoare asociate; generatoare de program; procesoare de testare a programelor; limbaje, compilatoare, translatoare; editoare de documentație; editoarele de texte etc.

Expresie a preocupărilor pe linia constituirii unor medii de programe și respectiv a unui cadru de producție care au ca efect trecerea la elaborarea pe principii industriale a programelor, o constituie organizarea *atelierelor de programare*.

La data realizării prezentului material în cadrul ITCI sînt în curs de desfășurare cercetări de elaborare a unui mediu de dezvoltare programe compatibil cu mediul UNIX (pe care-l vom descrie în paragraful următor).

Mediile evoluate destinate asistării de calculator a tuturor fazelor dezvoltării programelor cărora în plan organizatoric le corespund *atelierele de programare* au la bază pe lîngă *instrumentele software* de ajutor în programare o anumită tehnologie precum și o anumită concepție de organizare și conducere și o metodologie de desfășurare a procesului de elaborare a programelor.

În [116, 115] se prezintă un mediu de dezvoltare programe compus din:

— *un mediu de compoziție* care asistă programatorul în procesul proiectării software-ului; în acest context se definește un limbaj formal pentru programarea la scară largă, similar cu partea vizibilă a modulelor ADA;

— *un mediu de programare* constituit dintr-un set integrat de instrumente pe care programatorul individual le utilizează în procesul rafinării, de la descrierea algoritmilor pînă la obținerea unui program executabil;

— *un mediu de conducere* constituit dintr-un set integrat de instrumente care asigură comunicarea și cooperarea între programatorii unei echipe, prin generarea și gestionarea versiunilor și documentațiilor întregului sistem software.

---

*) La data realizării prezentului material.

17.2. Mediul de programare UNIX

Unul dintre cele mai cunoscute și utilizate medii de programare în lume este mediul de programare UNIX [108, 114] care are la bază sisteme de operare cu același nume.

Sistemul de operare UNIX, sistem interactiv în timp partajat, a devenit unul din cele mai răspîndite sisteme de operare datorită mai multor motive:

— posedă o interfață utilizator (limbaj de comandă) simplă și eficientă;

— asigură o mare varietate de utilitare software pentru dezvoltarea de programe;

— este un sistem portabil, asigurînd totodată și portabilitatea programelor scrise sub acest sistem.

Portabilitatea, ca unul din marele avantaje ale sistemului UNIX, rezultă din faptul că cea mai mare parte a monitorului sistemului și toate programele utilitare sînt scrise în limbajul de nivel înalt C. Transferul monitorului UNIX pe alt calculator presupune existența, pe noul calculator, a unui compilator pentru limbajul C și rescrierea unei mici porțiuni de cod dependentă de mașină. În schimb compilatoarele pentru alte limbaje, editoarele de texte și toate celelalte programe utilitare sînt scrise în limbajul C, deci transferul lor este imediat (odată ce există compilatorul C). În continuare vom enumera cîteva din programele existente în sistemul UNIX, ce asigură dezvoltarea de programe sub acest sistem și formează așa numitul mediu de programe UNIX.

17.2.1. Limbajul de comandă (shell)

Programul **shell**, care implementează limbajul de comandă, constituie interfața dintre utilizator și sistem. El decodifică cererile utilizatorului (introduse de la terminal) și le transmite monitorului. În momentul în care utilizatorul deschide o sesiune de lucru, de la terminal, programul **shell** afișează caracterul $ (urmat de un spațiu) invitînd utilizatorul să introducă diverse comenzi. După execuția unei comenzi, **shell** afișează $ arătînd că este gata să accepte comanda următoare. Acest ciclu continuă pînă la introducerea caracterului control-D, care înseamnă terminarea sesiunii de lucru (log out).

Aproape toate comenzile shell sînt de fapt nume de programe ce sînt lansate în execuție de către **shell** pentru a executa comanda respectivă. Astfel utilizatorul, prin crearea programelor sale, extinde setul de comenzi shell.

Formatul unei comenzi este următorul:

Comanda {argument} {argument} ... {argument}

Pentru execuția comenzii, **shell** caută mai întîi numele programului (dat de numele comenzii) în catalogul utilizatorului. Dacă nu este aici, îl caută în catalogul sistem (/**bin**); dacă nu-l găsește nici aici îl caută în extensia catalogului sistem (/**usr**/**bin**). Dacă nu există nici în această extensie se răspunde cu 'fișier inexistent'. De exemplu, compilatorul FORTRAN 77 se execută cu comanda:

$ **f77** nume fișier

La deschiderea unei sesiuni de lucru se asignează automat utilizatorului un fișier standard de intrare și un fișier standard de ieșire. În general acestea sînt terminalul de intrare. Foarte multe comenzi scriu sau citesc pe/din fi-

şierele standard şi de aceea ele nu se mai specifică în cadrul comenzii. De exemplu, listarea conţinutului catalogului utilizatorului se face cu comanda:

$ ls

iar lista se obţine pe terminalul de lucru.

Dacă se doreşte obţinerea listei pe alt dispozitiv sau într-un fişier se redirectează fişierul standard de ieşire. Exemplu:

$ ls > nume fişier

În acest caz lista se obţine în fişierul specificat. Dacă se doreşte adăugarea listei într-un fişier, fără a distruge informaţia precedentă, se utilizează comanda de redirectare prin ≫:

$ ls ≫ nume fişier.

Dacă utilizatorul doreşte lansarea comenzii următoare fără a aştepta terminarea execuţiei celei precedente, trebuie să informeze programul shell prin introducerea caracterului **&** la sfîrşitul comenzii precedente. Exemplu:

$ prog **&**
20
$ ed

Programul este lansat în execuţie, dar nu se aşteaptă terminarea lui. **Shell** cere introducerea noii comenzi prin afişarea caracterului $, iar utilizatorul cheamă editorul **ed**, ce va lucra în paralel cu prog. După lansarea programului prog se remarcă tipărirea unui identificator de program (20) ce poate fi utilizat pentru a controla execuţia lui prog. Exemplu:

$ kill 20

Se pot lansa în paralel oricîte programe (în funcţie de capacitatea de lucru a sistemului).

Există posibilitatea ca două programe să comunice direct între ele, în sensul că zona tampon de ieşire a unui program se depune în zona de intrare a altui program. Acest proces de înlănţuire automată (pipeline) se specifică prin caracterul '|'. Exemplu:

$ who | sort | lpr

Comanda **who** produce ca ieşire lista utilizatorilor existenţi în sistem. Această listă este preluată de programul **sort** (care o tipăreşte în ordine alfabetică) şi trecută apoi programului **lpr** pentru a fi listată pe imprimantă. Trebuie remarcat că nici un progam nu aşteaptă terminarea celui precedent. De exemplu, imprimanta începe să tipărească imediat ce programul **sort** i-a transmis prima linie.

Utilizatorul are posibilitatea să-şi creeze fişiere cu comenzi **shell**, ce se lansează cu comanda:

$ sh nume fişier

Fişierul specificat devine fişier standard de intrare iar următoarele comenzi shell vor fi citite din acest fişier şi executate secvenţial, pînă la epuizarea fişierului. Fişierul de comenzi poate fi executat în paralel cu comenzi de la terminal, dacă se specifică:

$ sh nume fişier **&**

Fișierele de comenzi pot conține parametri ce se înlocuiesc cu valorile specificate la lansarea în execuție. Există cel mult nouă parametri, ce se notează astfel: $1, $2, .., $9.

Exemplu: se dorește compararea fișierului 'file' cu o mulțime de alte fișiere. Comanda:

$ cmp file1 file2

permite compararea a două fișiere. Se creează un fișier de comenzi com:

$ com file $1

ce se apelează astfel:

$ sh com fis1

În acest caz se compară fișierul 'file' cu fis1.

Execuția unei comenzi din fișierul de comenzi poate fi condiționată de rezultatul execuției comenzii precedente. Aceasta este posibil prin utilizarea construcției condiționale **„if ... then ... else ... fi"**. Repetarea unor comenzi se poate face cu structura **„while ... do ... done"**.

Există unele caractere ce sînt interpretate de shell și desemnează acțiuni speciale. Astfel caracterul ';' este echivalent cu RETURN. În felul acesta se pot scrie mai multe comenzi într-o linie, despărțindu-le cu ';'. Caracterul * semnifică „orice șir de caractere", iar caracterul '?' înseamnă „orice caracter". De exemplu, dacă se dorește ștergerea tuturor fișierelor ce încep cu șirul de litere „pro" se va executa comanda:

$ rm pro*

În sistemul UNIX extensia (sau tipul) fișierului este în general de o singură literă (exemplu: prog.f, prog.a, prog.o). Dacă se dorește ștergerea tuturor fișierelor de nume prog, indiferent de tip se va utiliza comanda:

$ rm prog.?

De asemenea, se poate specifica un interval de caractere: orice caracter cuprins în acest interval dă naștere la o comandă validă. De exemplu, prin comanda:

$ rm prog.[a-1] [1, 2]

se șterg toate fișierele a căror extensie este o literă de la a la 1 sau cifra 1 sau 2.

Vom prezenta în continuare cîteva din cele mai importante comenzi shell. Intrarea în sistem, sau deschiderea sesiunii de lucru, se face cu comanda:

$ login [nume utilizator]

Sistemul răspunde pentru introducerea parolei. Un utilizator poate deschide o nouă sesiune de lucru, fără a fi necesar ca sesiunea curentă să fie închisă.

Ieșirea din sistem (sau închiderea sesiunii de lucru) se face prin introducerea caracterului control-D.

Comanda:

$ mail nume utilizator

permite transmiterea unui mesaj unui alt utilizator. Nu este nevoie ca utilizatorul specificat să aibă deschisă sesiunea de lucru. Sistemul depune mesajul într-o cutie poștală, iar la deschiderea sesiunii utilizatorul este anunțat de sosirea corespondenței. Pentru citirea mesajelor se introduce comanda:

$ mail

fără a specifica un nume de utilizator.

Comanda **mail** este urmată de mesajul de transmis. Sfîrșitul de mesaj se indică prin '.' sau control-D.

Exemplu:

$ mail *dana*
programul x.f mai are o eroare;
nu s-a corectat încă.

Transmiterea unor mesaje urgente, fără a mai fi depuse în cutia poștală se face cu comanda:

$ write nume utilizator [terminal]

Pentru a fi efectivă, utilizatorul trebuie să aibă sesiune deschisă la terminalul specificat. Sfîrșitul de mesaj se indică apăsînd control-D.

Dacă nu se dorește recepționarea de mesaje de la alți utilizatori se execută comanda:

$ mesg n

unde n semnifică „no". Validarea recepției de mesaje se face cu opțiunea „yes", adică:

$ mesg y

Comanda:

$ date

afișează la terminal data curentă și timpul.

Comanda:

$ time comandă

permite execuția comenzii specificate afișîndu-se și timpul de execuție al ei.

Execuția unui fișier de comenzi la o dată fixată în viitor se face cu comanda:

$ at timp [zi] fișier

Comanda:

$ man

permite listarea pe porțiuni (explicații) din manualul sistemului UNIX. De exemplu descrierea comenzii „date" se obține cu comanda:

$ man date

Comanda:

$ **cd** catalog

schimbă catalogul curent cu cel specificat.

Comanda:

$ **kill** proces

termină execuția procesului specificat. Procesul se identifică printr-un număr. Numărul de proces se poate afla cu comanda:

$ **ps** [opțiune] proces

unde „opțiune" poate fi „**a**" pentru a lista toate procesele asociate terminalului, „**l**" pentru a indica formatul lung (complet) de listare etc.

Prin comanda:

$ **lpr** [opțiune] ... [fișier] ...

se constituie coada fișierelor de listat la imprimantă.

Comanda:

$ **ls** [opțiune] catalog

listează conținutul (numele fișierelor) unui catalog.

Opțiunea poate fi:

— **l**: listare completă
— **s**: listare dimensiune fișiere (în număr de blocuri)
— **t**: listare în ordine cronologică, etc.

Prin comanda:

$ **mkdir** catalog

se creează un nou catalog.

Comanda:

$ **passwd**

schimbă parola sau definește o parolă nouă.

Comanda:

$ **size** fișier

afișează dimensiunea în octeți a fișierului specificat.

Comanda:

$ **wc** [opțiune] fișier

numără caracterele, cuvintele și liniile dintr-un fișier.

Prin comanda:

$ **who**

se afișează utilizatorii cu sesiune deschisă la sistem și terminalele respective.

Există, de asemenea, mai multe comenzi pentru manipularea fișierelor.

Comanda:

$ **cat** fișier1 fișier2

concatenează cele două fișiere și le listează la terminalul de ieșire. Concatenarea se poate face într-un alt fișier dacă se redirectează fișierul standard de ieșire către acel fișier.

Exemplu:

$ cat fișier1 fișier2 > fișier3

Copierea unui fișier în alt fișier se realizează cu comanda:

$ cp fișier1 fișier2

În acest caz fișierul va fi copiat în fișier 2.

Redenumirea unui fișier se face cu comanda:

$ mv fișier1 fișier2

Numele fișier1 devine fișier2.

Punerea în evidență a diferențelor dintre două fișiere se realizează cu comanda:

$ diff .[opțiune] fișier1 fișier2

Cele două fișiere se compară linie cu linie și se listează ce trebuie făcut ca cele 2 fișiere să devină identice. Opțiunea poate fi—b, de exemplu, pentru a ignora spațiile de la sfîrșit de linie.

Comanda:

$ comm [opțiune] fișier1 fișier2

lucrează pe fișiere sortate și listează pe trei coloane, respectiv: liniile ce apar numai în fișier1; liniile ce apar numai în fișier2; liniile ce apar în ambele fișiere.

Opțiunea poate fi cirfa 1, 2 sau 3 precedată de semnul —, ceea ce înseamnă suprimarea coloanei indicate.

Comanda:

$ cmp [opțiune] fișier1 fișier2

istează diferențele dintre 2 fișiere, indicînd numerele de linii diferite.

17.2.2. Sistemul de fișiere

Sistemul UNIX permite manipularea eficientă a structurilor de fișiere.

Fișierul, în mod normal, este văzut ca o sucesiune de octeți stocați pe disc sau alt suport magnetic. Nu există organizare logică specială pentru fișier. Sfîrșitul de fișier se marchează cu caracterul control-D. Fișierele sînt stocate pe disc în cataloage (sau directoare) ce permit regăsirea lor rapidă. Un catalog este de fapt un fișier ce conține nume de fișiere și adresa fizică a fișierelor pe disc.

Deoarece catalogul este un fișier se pot construi foarte simplu ierarhii de cataloage, adică un catalog să conțină numele unui alt catalog. În principiu există un catalog rădăcină, ce conține numele tuturor cataloagelor asignate utilizatorilor. În catalogul său, utilizatorul își poate crea subdirectoare, structura devenind arborescentă. În felul acesta utilizatorul manipulează numai fișierele sale, aflate în catalogul său, fără a avea nevoie să știe dacă există pe același disc fișiere ale altor utilizatori.

Accesul la un fișier se face specificînd drumul în ierarhia de cataloage pe care se ajunge la el, pornind de la catalogul rădăcină:

$catalog_0/catalog_1/catalog_2/ \ldots /$numefișier

La deschiderea sesiunii, catalogul utilizatorului devine catalog curent și deci drumul de acces de la catalogul rădăcină la cel al utilizatorului nu mai trebuie specificat. Numai subcataloagele utilizatorului trebuie specificate pentru accesul la un fișier. De cele mai multe ori catalogul în care se lucrează este catalog curent, deci numele fișierului este suficient.

Drumul pînă la catalogul curent, în ierarhia de cataloage, se poate afișa cu comanda:

$ pwd

care tipărește:

/cat1/cat2/ ... /catalog curent

Numele catalogului rădăcină nu există, deci listarea drumului de cataloage începe cu / după care urmează numele cataloagelor create de sistem.

Schimbarea de catalog se face cu comanda:

$ cd /cat1/cat2/ ... / catn

prin care catn devine catalog curent. Dacă utilizatorul lucra în cat2 de exemplu, ce era catalog curent, se poate specifica drumul de cataloage începînd cu următorul catalog după cat2.

Fișierele sînt protejate contra acceselor nepermise. Există 3 forme de acces la fișier: citire, scriere și execuție. Ele pot fi combinate pentru a da naștere la diferite posibilități de acces. Există trei categorii de utilizatori: posesorul fișierului, membrul unui grup de utilizatori și publicul. Fiecare utilizator din sistem are un nume și un număr de grup. Clasificarea în grupuri este necesară pentru elaborarea unor aplicații complexe, în care se disting activități relativ independente. La crearea unui fișier se asignează, prin lipsă, următoarele drepturi de acces: toate drepturile de acces pentru posesor, citire, și execuție pentru grup, execuție doar pentru public. Posesorul fișierului poate schimba cheile de protecție ale fișierului prin comanda:

$ chmod utilizator drepturi numefișier

unde:

— utilizatorul poate fi:

—u: posesorul (user)
—g: grup (group)
—o: public (others)
—a: toți (all)

— drepturi de acces:

+r: citire permisă, —r: citire nepermisă
+w: scriere permisă, —w: scriere nepermisă
+x: execuție permisă, —x: execuție nepermisă.

Structura de cataloage a sistemului UNIX este (identarea specifică structura ierarhică):
catalog rădăcină, ce conține:

dev: fișiere speciale pentru dispozitive fizice
bin: programe utilitare, compilatoare (în format executabil)
lib: biblioteci sistem
etc: programe privilegiate pentru administratorul de sistem
tmp: fișiere temporare pentru diverse utilitare
usr: cataloage utilizator, ce conține:
  bin: utilitare mai puțin folosite
  tmp: fișiere temporare
  dict: dicționarul de cuvinte:
  lib: biblioteci mai puțin folosite
  man: manualele sistemului
  spool: programe de listare
  user-1: catalog pentru utilizatorul 1
  user-2: catalog pentru utilizatorul 2
  .....
  user-n: catalog pentru utilizatorul n

Cuvintele user-1, user-2, ..., user-n sînt de fapt numele de înregistrare al utilizatorilor în sistem (Ionescu, Dan etc.).

Regăsirea drumului de acces la un fișier se face cu comanda:

$ find catalog condiții

unde catalog indică de la care catalog să se demareze căutarea fișierului iar condiții indică ce trebuie căutat. Catalogul curent se notează cu '.' iar catalogul imediat superior lui se notează cu '..'. De exemplu prin comanda:

$ find . —name 'fișier' **—print**

se caută și se tipărește la terminal (print) lanțul de cataloage ce pornește de la catalogul curent (.) pînă la catalogul ce conține fișierul de nume (name) 'fișier'.

Structura de cataloage se creează pe suporturi magnetice, de exemplu de tip disc, cu comandă:

$ mkfs fișierespec dimensiune

unde fișierspec este numele fișierului special ce servește accesul la dispozitivul fizic, iar dimensiune este un număr zecimal ce specifică numărul de blocuri alocat pentru structura de fișiere. Un disc cu structură de fișiere nu devine disponibil utilizatorului decît după comanda de montare:

$ mount /dev/xxx numefișier

unde /dev/xxx este fișierul special ce manipulează perifericul respectiv, iar numefișier se substituie catalogului rădăcină.

Din punct de vedere a structurii interne, fișierul este o colecție de octeți de 8 biți, ce reprezintă caractere sau valori numerice. Octeții pot fi grupați în șiruri de lungime variabilă numite articole. Un caracter special delimitează articolele. Sfîrșitul de fișier este indicat de caracterul control-D. Există programe UNIX care lucrează cu fișiere cu o structură logică bine definită.

Sistemul UNIX asigură accesul secvențial la fișiere. Pentru programele scrise în limbajele de nivel înalt, ca FORTRAN sau C, se asigură și accesul direct la fișiere.

Operațiile de intrare-ieșire (read/write) sînt bufferizate, datorită diviziunii fișierului pe suportul magnetic în blocuri de 512. octeți.

Sistemul UNIX este prevăzut cu fișiere speciale, numite biblioteci, ce sînt înregistrate în catalogul **lib**. Ele sînt consultate de unele programe utilitare (de exemplu, editorul de legături, **ld**) fără o indicație specială din partea utilizatorului. Aceste biblioteci conțin, în general, rutinele comune pentru operațiile de intrare-ieșire, funcții matematice etc. Utilizatorul își poate crea biblioteci proprii utilizînd programul **ar**.

17.2.3. Limbaje de programare

17.2.3.1. *Limbajul C*

Limbajul C este limbajul de bază al sistemului UNIX. Structura acestui limbaj se situează între limbajele FORTRAN și PASCAL, cu diferența că limbajul C este mult mai aproape de mașină decît celelalte două limbaje: lucrul cu adrese, indecși, operatori de incrementare și decrementare ce sînt caracteristici limbajelor de asamblare.

Un program C constă din una sau mai multe funcții, ce posedă sau nu posedă parametri. Ele se declară prin nume urmat de lista de argumente:

Exemplu:

funct (a, b, c) — funcție cu argumente

funct ( ) — funcție fără argumente (parantezele sînt obligatorii)

Tipul variabilelor argument se declară imediat după paranteza ce termină lista de argumente, fiecare declarație fiind terminată de caracterul;. Codul funcției se închide între paranteze de tip acoladă ({ }).

Exemplu: **funct** (a, b, c) **int** a; **float** b, c; {... instrucțiuni C ...}

Prin lipsă, tipul funcției se consideră întreg **(int)**. El se poate însă declara. De exemplu:

double funct (a, b, c) **int** a, b, c; {...}

În acest caz se declară o funcție de tip întreg cu dublă precizie pentru calcule și rezultate.

Funcția de bază a programului, care primește controlul la lansarea în execuție, se desemnează cu cuvîntul cheie „main". Nu există alt cuvînt cheie pentru desemnarea unei funcții. Dacă un nume este urmat de paranteze se va considera automat ca declarație de funcție. De asemenea, ieșirea din funcție are loc la întîlnirea acoladei de terminare al codului funcției (}). Instrucțiunea „return" se utilizează doar pentru a specifica valoarea funcției, ce se asociază numelui funcției.

Funcțiile în C pot fi recursive, dar nu se admite declararea unei funcții în corpul altei funcții. Variabilele se transmit funcției prin valoare și nu prin nume, deci nu se distruge conținutul inițial al variabilei, dinainte de apel. Dacă se dorește modificarea variabilei cu rezultatul calculat de funcție se folosește un mecanism de apel prim pointeri, asemănător apelului prin nume.

Tipurile de date disponibile în limbajul C sînt următoarele:

— **int** — întreg cu semn (un cuvînt de 16 biți),
— **char** — caracter (un octet de 8 biți),
— **short** — întreg scurt (1 cuvînt de 16 biți),
— **long** — întreg lung (1 cuvînt de 32 de biți),
— **unsigned** — întreg fără semn (1 cuvînt de 16 biți),
— **float** — real (două cuvinte (32 biți)),
— **double** — precizie dublă (4 cuvinte (64 biți)).

Lungimea (în biți) pentru fiecare tip de dată s-a specificat în cazul unui minicalculator cu cuvîntul de 16 biți.

Deși nu există tipul de dată „logic" sau „boolean", toate valorile diferite de zero semnifică „true", iar valoarea zero înseamnă „false".

Utilizatorul trebuie să declare tipul oricărei variabile utilizate în programul său. Declarația se face specificînd tipul urmat de numele variabilei (sau lista de variabile separate prin virgulă). Exemplu:

```
int a, b, c;
float x;
```

O variabilă externă funcției se declară prin cuvîntul cheie „extern". Exemplu:

```
funct (a, b) float a, b; {extern int i, j, k; ...}
```

O variabilă pointer conține adresa unei alte variabile.

Un pointer se desemnează prin caracterul „*" iar adresa unei variabile prin caracterul „&". Ex.:

```
float *p
p = &x
```

În acest caz p pointează pe variabila x.

Un tablou se declară specificînd mai întîi tipul, apoi numele tabloului, urmat de numărul de dimensiuni. Exemplu:

```
int tab [10] [10]
```

Aceasta este o declarație a tabloului tab de două dimensiuni cu 10×10 elemente.

Constantele de tip caracter se reprezintă între ghilimele simple. Exemplu: 'x'. Constantele șir se cuprind între ghilimele duble. Exemplu: "program".

Limbajul C este prevăzut cu un set bogat de operatori unari, binari, relaționali și de asignare.

Operatorii unari sînt: * — indirectare (valoarea unui pointer reprezintă o adresă); & — creare de pointer (adresă variabilă); — — negare aritmetică (valoare negativă a valorii variabilei); ! — negare logică (zero produce 1, o valoare diferită de zero devine 0); ++ — incrementarea variabilei specificate; — — — decrementarea variabilei specificate; **sizeof** — numărul de caractere în variabila specificată; **(type)** — conversie tip.

Acești operatori preced numele variabilei, cu excepția operatorilor ++ și — — care pot fi înainte sau după variabilă după cum se dorește incrementarea/decrementarea valorii variabilei înainte sau după utilizarea ei.

Operatorii binari sînt: * — înmulțire; / — împărțire; % — calculul restului al unei împărțiri întregi; + — adunare; — — scădere; ≪ sau ≫ — se decalează primul operand spre stînga (sau spre dreapta) cu un număr de biți indicat de operandul al doilea; & — și boolean (bit cu bit); SAU boolean (bit cu bit); ↑ SAU exclusiv (bit cu bit); && — ȘI logic (produce 1 dacă ambii operanzi sînt diferiți de zero, sau zero în caz contrar); || — SAU logic (produce 1 dacă cel puțin unul din operanzi este nenul, sau zero dacă ambii sînt nuli)

Operatorii de relație sînt: < — mai mic; > — mai mare; <= — mai mic sau egal; >= — mai mare sau egal; == — egal; != — diferit.

În general operatorii binari și de relație se utilizează în formatul: operand1 operator operand2. Operatorul „=" se poate folosi pentru a indica o operație asupra aceleiași variabile.

De exemplu:

a *= 5; b —= 3

semnifică:

a = a * 5; b = b — 3.

În același sens se pot folosi notațiile: *=, /=, %=, ≪=, ≫=, &=, !=.

Limbajul C permite declararea și manipularea structurilor de date. Declararea lor se face prin cuvîntul cheie „struct" urmat de numele structurii, după care se descrie, între acolade, tipul de date ce constituie structura. Exemplu:

struct persoană {**char** nume [15], prenum [15], **int** vîrsta}

Alte structuri de același tip se vor declara prin specificarea numelui structurii urmat de „'.'" și de numele elementului din structură. Exemplu:

i = persoană.vîrstă
persoană.vîrstă = j

În principiu limbajul C nu posedă instrucțiuni de intrare-ieșire. În schimb biblioteca standard de funcții conține rutinele necesare pentru a efectua operații formatate de intrare-ieșire. Ele se pot introduce în programul C prin instrucțiunea:

include <stdio.h>

Imprimarea cu format se face cu funcția **printf**, iar introducerea de date cu format se realizează în funcția **scanf**.

Compilatorul de C lucrează în mai multe faze:

— un preprocesor analizează și rezolvă instrucțiunile **#include** și **#define**,
— se generează apoi un cod intermediar C,
— se generează limbajul de asamblare (specific mașinii).

Utilizatorul apelează apoi asamblorul pentru a produce codul obiect al programului.

Editorul de legături (apelat de utilizator sau în continuare de către asamblor) creează codul executabil al programului.

17.2.3.2. *Limbajul FORTRAN 77*

FORTRAN 77 este unul din ultimele standarde ale limbajului FORTRAN. În UNIX compilatorul de FORTRAN 77, cunoscut sub numele **f77**, produce la ieșire cod intermediar C, ca de altfel toate celelalte limbaje de înalt nivel.

Acest cod este transformat în limbaj de asamblare de către faza 2 a compilatorului C, apoi în cod mașină de către asamblor. Prin înlănțuirea automată a execuției diverselor programe (pipe line) acest proces nu este sesizat de utilizator, care la comanda:

$ f77 numefișier

obține în final un program executabil (în fișierul a.out).

Limbajul FORTRAN 77 din sistemul UNIX prezintă cîteva extensii față de standardul 77, și anume:

a) Instrucțiunea se poate plasa oriunde în linia sursă și poate depăși limita de 72 de caractere. Continuarea unei instrucțiuni pe linia următoare se indică prin caracterul **&** în prima coloană a liniei de continuare. Eticheta se desparte de instrucțiune prin caracterul de tabulare orizontală (tab).

b) Compilatorul recunoaște și setul de litere mici, astfel încît var și Var devin variabile distincte. Cuvintele cheie se recunosc numai în scrierea cu litere mici.

c) — Se recunosc unele caractere speciale, fiind precedate de caracterul \, și anume: \n — salt la linie nouă; \t — tabulare orizontală; \f — salt la nouă pagină etc.

d) Șirurile de caractere pot fi cuprinse între ghilimele simple sau duble. Astfel, dacă un apostrof apare în cadrul șirului, atunci se pot folosi ghilimele duble pentru a delimita șirul.

Exemplu:

„luna august, anul '84"

e) Apar două noi tipuri de date:

integer*2 — ocupă 2 octeți

double complex — ocupă de 4 ori mai mare spațiu decît pentru variabilele de tip real

f) Se poate invalida convenția FORTRAN prin care se stabilește prin lipsă tipul unei variabile (sînt de tip întreg toate variabilele ce încep cu literele i, j, k, l, m sau n). În acest scop se folosește instrucțiunea:

implicit undefined (a—z)

În felul acesta toate variabilele trebuie să aibă tipul declarat explicit, ceea ce asigură eliminarea unor erori legate de conversia datelor.

g) Se poate însera text în programul FORTRAN (cod sursă) dintr-un fișier ce se specifică în instrucțiunea „include":

#include nume-fișier

17.2.3.3. *Limbajul RATFOR*

RATFOR (Rational FORTRAN) reprezintă o extensie a limbajului FORTRAN, incluzînd multe din structurile de control din PASCAL și C. Nu există compilator de RATFOR ci doar un preprocesor care traduce programul RATFOR în FORTRAN standard. Fazele prin care trece un program RATFOR pentru a ajunge în instrucțiuni mașină sînt:

sursă RATFOR → sursă FORTRAN → cod intermediar C → limbaj de asamblare → cod obiect.

Execuția diverselor faze este înlănțuită (pipe line) fără intervenția utilizatorului.

Formatul de scriere al instrucțiunilor RATFOR este liber, în sensul că instrucțiunile se pot plasa oriunde în liniile sursă și se pot continua pe oricîte linii fără a utiliza caractere speciale de continuare. În general se utilizează caracterul „';'" pentru a separa instrucțiunile de pe aceeași linie (este opțional pentru ultima instrucțiune de pe linie).

Etichetele (numere) se pot plasa oriunde în linia sursă dar înaintea instrucțiunii pe care o marchează. Ea va fi plasată de preprocesor în spațiul coloanelor 1—5.

Operatorii de comparare capătă în RATFOR o formă mai apropiată de cea utilizată în matematică. De exemplu: .gt. devine >; .lt. devine <; .ge. devine >= etc.

Liniile ce conțin caracterul % în prima coloană nu sînt prelucrate de preprocesorul RATFOR; ele se copiază întocmai în fișierul de ieșire FORTRAN.

Așa cum am mai spus structurile de control din RATFOR sînt asemănătoare cu cele din PASCAL și C. Astfel blocul de instrucțiuni se indică prin paranteze de tip acoladă ({ }). Instrucțiunea **if** acceptă ca acțiune un bloc de instrucțiuni. Exemplu:

if (i > j) {x = y; z = z + 1}

Instrucțiunea de selecție **if** admite și partea **else**. Exemplu:

if (x < 0) **then** a = 0 **else** a = 1.

Structura repetitivă se poate realiza în patru moduri. Exemple:

do i = 1,100 {grup de instrucțiuni}
for (i = 2 ; i = r − 1 ; i = r) {grup de instrucțiuni}
while (i > 0) {grup de instrucțiuni}
repeat {grup de instrucțiuni} **until** (i > 0)

În grupul de instrucțiuni se pot folosi două instrucțiuni, **next** și **break**, relative la execuția procesului interativ.

Instrucțiunea **next** provoacă reluarea ciclului fără a mai executa instrucțiunile ce urmează după ea. Instrucțiunea **break** provoacă ieșirea necondiționată din ciclu.

Limbajul RATFOR permite substituirea de text sau excluderea de text. Instrucțiunea **define** permite asignarea unui șir de caractere unui nume. De exemplu instrucțiunea:

define param 10

are drept consecință înlocuirea (în textul sursă FORTRAN generat de RATFOR) a simbolului param cu șirul de caractere 10.

Instrucțiunea **include**, similară cu cea din FORTRAN 77, inserează text dintr-un fișier extern programului RATFOR.

Formatul ei este:

include nume-fișier

Un dezavantaj al limbajului RATFOR îl constituie faptul că toate cuvintele cheie ale limbajului sînt rezervate (ceea ce nu este cazul pentru limbajul FORTRAN standard).

De exemplu declarația:

integer if

nu este permisă.

Preprocesorul RATFOR poate fi apelat direct:

$ **ratfor** nume-fișier

sau de către compilatorul **f77**. În acest ultim caz numele fișierului specificat în linia de comandă pentru **f77** trebuie să aibă extensia 'r'. Exemplu:

$ **f77** prog.r

unde prog este numele fișierului cu linii sursă RATFOR.

17.2.3.4. *Limbajul BASIC*

Sistemul UNIX posedă un interpretor al limbajului BASIC, ce recunoaște un subset de instrucțiuni BASIC. Se consideră a fi util pentru calculatoare cu resurse de calcul foarte reduse (calculatoare de birou, personale etc.).

Instrucțiunile BASIC sînt scrise cu litere mici. Numele de variabile trebuie să înceapă cu o literă, urmată de litere și cifre, și pot fi oricît de lungi (pentru majoritatea versiunilor de BASIC, doar primele 4 caractere sînt semnificative; astfel 'persoana' și 'pers' reprezintă aceeași variabilă).

Operatorii recunoscuți sînt cei aritmetici și relaționali, la care se adaugă operatorii logici & (și) și | (sau).

O instrucțiune constă într-o expresie care poate include sau nu operatorul de asignare (=). Dacă expresia nu conține operatorul de asignare se evoluează imediat și se afișează rezultatul.

Dacă instrucțiunile sînt prefixate de un număr se memorează și se vor executa la comanda **run**. Programul poate fi examinat cu comanda **list**.

În momentul în care se apelează interpretorul BASIC, prin comanda:

$ **bas** nume-fișier

acesta va începe să citească din fișierul specificat, pînă la epuizarea lui, după care așteaptă comenzi de la terminalul de intrare.

Editarea fișierului specificat se poate face cu comanda **edit**. În acest caz interpretorul BASIC cere execuția editorului de texte **ed** și deci se pot folosi toate facilitățile acestuia pentru editarea programului basic.

Citirea unei date de la terminal, în timpul execuției programului, se realizează cu ajutorul funcției **expr** ( ). Exemplu:

x = expr ( )

Această instrucțiune are ca efect introducerea unui număr de la terminal și asignarea lui variabilei x.

17.2.3.5. *Limbajul PASCAL*

Există numeroase variante de compilatoare și interpretoare PASCAL sub UNIX. Versiunea Berkely, descrisă aici, este probabil cea mai răspîndită. Ea permite atît interpretarea cît și compilarea programelor PASCAL, fiind de fapt un subsistem PASCAL.

Există 5 programe ce formează subsistemul PASCAL și anume: **pi, px, pc, pxp** și **pxref**.

pi — realizează translatarea sursei PASCAL în cod intermediar iar **px** citește și interpretează acest cod intermediar (traduce codul intermediar în cod mașină, instrucțiune cu instrucțiune, executînd imediat instrucțiunea după traducerea ei). Prin comanda **pix** se realizează execuția înlănțuită a celor două programe.

pc — este un compilator ce produce cod executabil sau poate crea și module în limbaj de asamblare, ce se pot lega cu alte module scrise în alte limbaje de programare.

pxp și **pxref** — sînt utilitare pentru dezvoltarea de programe în PASCAL. **pxp** indică de cîte ori s-a executat fiecare instrucțiune iar **pxref** este un generator de referințe încrucișate, foarte util pentru localizarea variabilelor în aplicații de programe complexe.

Prin convenție fișierele sursă PASCAL posedă extensia p (de exemplu: prog.p). Fișierul de ieșire al interpretorului este **obj** iar programul executabil peste **a.out.**

Interpretorul de PASCAL (**pi**, **px**) este foarte util pentru dezvoltarea de programe în PASCAL. Astfel pentru depanarea programelor se asigură un vidaj post-mortem, contorizează numărul de instrucțiuni executate, iar dacă s-a depășit o anumită limită oprește execuția programului (evitîndu-se ciclurile infinite). De asemenea, contorizează și timpul de execuție al programului.

Suportă instrucțiunea include pentru inserarea de cod sursă extern programului:

 # include „nume-fișier.i";

În cazul unui minicalculator cu cuvintul de 16 biți, interpretorul limitează dimensiunea programului la 64 Kocteți, ceea ce înseamnă între 1000 și 2000 de linii sursă PASCAL.

Compilatorul de PASCAL (**pc**) produce cod executabil sau limbaj de asamblare. În general se utilizează pentru programe ce au fost depanate cu interpretorul **pi**. Există posibilitatea de a compila separat funcții sau proceduri ce vor fi, mai tîrziu, legate împreună.

Utilitarul **pxp** este foarte util pentru testarea programelor PASCAL. Prin afișarea numărului de execuții al fiecărei instrucțiuni se pun în evidență părțile testate din program și se obțin, deasemenea, informații prețioase pentru optimizarea programului prin reducerea codului frecvent executat.

17.2.3.6. *Asamblorul (as)*

În limbajul UNIX limbajul de asamblare este practic înlocuit de limbajul C. El se păstrează însă din necesitatea de a programa operații foarte apropiate de structura hardware a calculatorului, care nu pot fi codificate eficient în limbajul C.

17.2.3.7. *Editorul de legături (ld)*

Editorul de legături **ld** (linking loader) este, practic, transparent utilizatorului, pentru că majoritatea acțiunilor (compilări) introduc automat ca fază finală editarea de legături. În general este apelat în mod explicit pentru a reloca mai multe module rezultate din compilări diferite. Linia de comandă este:

$ **ld** fișier1, fișier2, ..., fișiern

Fișierul de ieșire al editorului de legături este, prin lipsă, **a.out** sau se poate numi explicit de către utilizator.

Bibliotecile sistem nu se specifică în cadrul comenzii, fiind automat consultate de către editorul de legături. Bibliotecile utilizator trebuie însă numite explicit.

17.2.4. **Prelucrarea textelor**

Sistemul UNIX se caracterizează și prin faptul că pune la dispoziția utilizatorilor un set bogat de utilitare pentru pregătirea, editarea și formatarea textelor. Pentru editarea textelor cele mai cunoscute programe sînt **ed** și **vi**. Pentru pregătirea și formatarea textelor, în sensul realizării de documentație, există următoarele programe: **nroff** (cu ieșirea pe terminal sau imprimantă), **troff** (cu ieșirea pe mașini fotografice speciale), **neqn** (pentru documentație matematică), **tbl** (pentru editarea de tabele), **spell** (pentru verificarea documentației) etc.

Există, de asemenea, și alte programe pentru compararea de texte, sortare, testare etc. utile în diverse aplicații.

17.2.4.1. *Editorul de texte (ed)*

Editorul de texte (**ed**) posedă un set simplu de comenzi, care permit citirea, modificarea, copierea unui text. Citirea se face dintr-un fișier, iar ieșirea poate fi în același fișier sau alt fișier. Este orientat pe linii, adică textul este văzut ca o colecție de linii sursă. Editorul face o copie a fișierului de intrare într-o zonă tampon (buffer) din memorie, numită zonă de editare sau zonă text. Utilizatorul poate examina liniile din această zonă, le poate modifica (sau corecta), după care rescrie zona în fișierul de ieșire. Dacă fișierul este scurt, zona tampon de editare poate intra în întregime în memoria fizică. În acest caz toate operațiile au loc în memorie. Dacă fișierul este prea mare și nu poate fi cuprins în întregime în memorie, atunci se alocă o parte din zona de editare în memorie, iar restul în fișier temporar pe disc. Gestiunea zonei astfel împărțită este făcută în întregime de către editor astfel încît utilizatorul nu are nici o cunoștință de transferurile disc-memorie Utilizatorul trebuie să precizeze, prin comandă editor, momentul în care dorește rescrierea zonei de editare în fișierul de intrare sau în alt fișier. Dacă se iese din editor fără a da această comandă, toate modificările executate anterior vor fi pierdute.

Editorul menține un pointer de linie ce precizează linia curentă în lucru. Linia curentă se identifică prin '.' iar ultima linie în fișier cu caracterul '$'.

Formatul uzual al comenzilor editorului este următorul:

[număr linie, [număr linie]] comandă

Toate comenzile editorului se reduc la o singură literă. De exemplu, listarea tuturor liniilor din zona de editare se face cu comanda:

1, $ p

unde 1 este numărul primei linii, $ specifică ultima linie iar p înseamnă „tipărește" (print).

Alte comenzi mai utilizate sînt:

a (append) — adaugă linii noi,
c (change) — modifică liniile specificate,
d (delete) — șterge liniile specificate,
f (filename) — tipărește numele fișierului din zona de editare,
g (global) — extinde aria noilor comenzi la toată zona de editare,
i (insert) — inserare de noi linii,
j (join) — formează o singură linie din două,
m (move) — transferă linii,
n (number) — afișează liniile cu numerele asociate lor,
p (pointer) — poziționează pointer-ul de linii și afișează liniile,
q (quit) — se iese din editor,
r (read) — citește un fișier în zona de editare,
t (transfer) — transferă linii dar nu le șterge din locul original,
w (write) — scrie zona de editare într-un fișier,
W (write) — adaugă zona de editare unui fișier,
= — afișează valorile numerice ale liniilor . și $.

Dacă în comandă nu se precizează nici un număr de linie se va considera linia curentă pentru execuția comenzii.

La comenzi eronate editorul răspunde cu unicul mesaj de eroare, și anume „?".

Dacă lipsește parametrul „comandă" din formatul comenzii prezentat anterior, se va considera implicit, comanda p. De exemplu, tipărirea liniei 6 din textul sursă se poate face fie cu comanda:

6 p

fie introducînd numai numărul liniei:

6

Dacă se dorește afișarea liniei a treia după linia curentă se introduce comanda:

+3

care reprezintă o formă redusă a comenzii:

.+3 P

unde . specifică pointerul pentru linia curentă.

În același fel se poate folosi și operatorul „`—´" pentru a reveni înapoi.

Prin comanda i se inserează noi linii sursă. De exemplu:

```
i
------
linia n
linia n + 1
.
```

Ieşirea din modul de inserare linii se face introducînd un . ca prim caracter (şi unicul) pe o linie nouă. Linia curentă devine ultima linie introdusă.

Prin comanda `a´ se introduc noi linii, dar după linia curentă (comanda i inserează textul înaintea liniei curente). Astfel, există posibilitatea de a adăuga text după ultima linie (comanda a) sau la început de fişier (comanda i).

Comanda de ştergere linii operează ca şi comanda p. Ex.

```
d          — şterge linia curentă
4,6 d      — şterge liniile 4, 5, 6 din fişierul sursă
—2, +2 d   — şterge linia curentă, plus 2 înainte şi după linia curentă
10,$ d     — şterge toate liniile începînd de la linia 10.
```

Prin comanda c se pot înlocui mai multe linii cu altele noi. Ex.:

```
—2,+2 c
linia n
linia n+1
.
```

Se înlocuieşte linia curentă împreună cu două linii mai sus şi două mai jos cu liniile n, n+1 etc. Sfîrşitul grupului noilor linii se indică prin caracterul „."

Liniile se pot identifica, nu numai prin număr, ci şi prin şirurile de caractere pe care le conţin. Exemplu, comanda:

```
/registru/
```

caută prima linie ce conţine şirul de caractere „registru", linia devenind linie curentă. Dacă se ajunge la sfîrşit de fişier se continuă căutarea de la începutul fişierului pînă la fosta linie curentă. Dacă nu se găseşte nici o linie cu şirul de caractere specificat editorul răspunde cu ?.

Substituirea de şiruri se face cu comanda s. De exemplu, comanda:

```
2,10 s /a — c/a — b/
```

înlocuieşte şirul a — c cu a — b în zona de linii cuprinsă între a doua şi a zecea linie.

Comanda:

```
s /xyt/xyz/ g p
```

provoacă înlocuirea şirului xyt cu xyz în tot fişierul (g) şi tipărirea liniilor modificate (p).

Comanda:

```
1,5 m +3
```

are ca rezultat transferarea primelor 5 linii după a 3-a linie după linia curentă.

Există comenzi pentru manipularea de fișiere, r (read), w (write), e (edit). Comanda r adaugă un nou fișier în zona de editare. Exemplu:

r fișier2
r fișier3

Se adaugă în zona de editare, fără a distruge conținutul anterior, două fișiere, ce devin practic un singur fișier.

Prin comanda w se scrie zona de editare într-un fișier specificat. Se poate indica doar o parte din zona de editare. De exemplu:

w fișier5
1,20 w fișier6

Prin comanda e se șterge zona de editare și se rescrie cu fișierul specificat. Exemplu:

e fișier7

Această comandă este echivalentă cu comenzile:

1,$ d — șterge toate liniile din zona de editare
r fișier7 — citește fișier 7 în zona de editare.

Comanda e salvează numele fișierului de intrare, iar dacă utilizatorul a uitat acest nume îl poate regăsi cu comanda f. Dacă fișierul specificat în comanda e nu există se creează un nou fișier, cu o singură linie, fără caractere.

Comanda q termină lucrul cu editorul. Înainte de a executa această comandă utilizatorul trebuie să scrie zona de editare în fișierul de ieșire cu comanda w.

17.2.4.2. *Editorul de ecran (vi)*

Acest editor se folosește pentru terminalele cu ecran (display-uri), fiind un utilitar foarte puternic, ce posedă mai mult de 100 de comenzi.

Ideea de bază de funcționare este similară cu cea a editorului **ed**: fișierul de intrare este copiat în zona de editare, dar tehnica de afișare este diferită și anume ecranul terminalului constituie o fereastră ce afișează o parte din zona de editare. Dimensiunea ferestrei este variabilă și este definită de utilizator (ea este limitată de dimensiunea ecranului). Fereastra display-ului poate fi deplasată pentru a afișa diferite porțiuni din zona de text. Deplasarea se poate face înainte sau înapoi prin următoarele comenzi:

control-F — deplasarea înainte, cu dimensiunea ferestrei
control-B : deplasare înapoi, cu dimensiunea ferestrei
control-D : deplasare parțială înainte (jos)
control-U : deplasare parțială înapoi (sus)

Pentru deplasarea parțială a ferestrei se specifică numărul de linii înaintea comenzii. La fel, pentru deplasarea totală a ferestrei se poate specifica un număr înaintea comenzii, ce constituie noua dimensiune a ferestrei. Aceste numere se memorează și rămîn valide și pentru următoarele comenzi de deplasare parțială, respectiv totală.

Linia curentă este indicată de un cursor (un caracter special) sau apare în video inversată etc. Deplasarea cursorului se face cu comenzi speciale:

h : deplasare cu un caracter spre stînga,
j : deplasare cu un caracter în jos,
k : deplasare cu un caracter în sus,
l : deplasare cu un caracter spre dreapta,
w : deplasare înainte, în linie, la cuvîntul următor,
b : deplasare înapoi, în linie la cuvîntul precedent,
H : poziționare cursor pe prima linie,
L : poziționare cursor pe ultima linie;
M : poziționare cursor în mijlocul ecranului.

Comanda 5H, de exemplu, înseamnă deplasarea cursorului pe a cincea linie din partea de sus a ecranului.

Majoritatea comenzilor editorului vi se reduc la un singur caracter, precedat sau urmat de valori numerice.

În cazul în care se dorește revenirea la situația dinainte de execuția unei comenzi incorecte se introduce comanda u ce restaurează zona de editare la starea anterioară execuției comenzii.

Prin comenzile i și a se inserează text chiar și în interiorul unei linii, dacă poziția cursorului este în interiorul liniei. Comanda i introduce text înaintea cursorului, iar comanda a — după cursor. De exemplu, comanda:

i program <esc>

introduce șirul 'program' imediat înaintea cursorului. Introducerea de mai multe linii se face apăsînd tasta RETURN în loc de ESCAPE.

Ștergerea unei linii se face cu comanda d. Alte comenzi de ștergere sînt codificate prin diverse litere ce urmează după d. Exemplu:

dw — șterge următorul cuvînt din linie,
d3w — șterge următoarele trei cuvinte din linie,
d3b — șterge trei cuvinte înapoi.

Comanda c înlocuiește șirul de caractere cu alte șiruri. Exemplu:

cw exemplu < esc >

înlocuiește următorul cuvînt cu șirul „exemplu".

Lansarea în execuție a editorului vi se face cu comanda:

$ vi nume-fișier

unde nume-fișier este specificatorul fișierului de editat.

Ca și în cazul editorului ed, copierea zonei de editare într-un fișier trebuie să se facă explicit. Comanda de scriere este:

: w < esc >

unde caracterul „':'" este necesar pentru a deosebi comanda de scriere de comanda w (cuvînt).

17.2.4.3. *Formatarea textelor (nroff)*

Programul **nroff** se utilizează pentru formatarea textelor în sensul punerii în pagină pentru elaborarea de rapoarte, documentații etc. **nroff** citește fișierul de intrare linie cu linie, prelucrează linia și o scrie în alt fișier reformatată conform unor comenzi.

Comenzile **nroff** încep în coloana 1 a liniei și sînt precedate de caracterul „.'." De exemplu, comanda:

.pl 60

stabilește dimensiunea paginii la 60 de linii. Deci după fiecare 60 de linii scrise în fișierul de ieșire se face salt la pagina următoare.

Comanda:

.11 62

stabilește numărul de caractere pe linie (62 de caractere).

Fiecare comandă este formată din 2 caractere și poate fi urmată de numere cu semn (+ sau —). De exemplu:

.11 + 2

stabilește lungimea liniei cu 2 caractere mai mare.

Punerea în pagină a liniilor (aranjarea la marginea dreaptă a paginii) se face de către **nroff**. Utilizatorul își introduce textul (cu editorul de texte) în care sînt inserate comenzi nroff pentru stabilirea formatului de ieșire. Alinierea la marginea dreaptă a paginii se face prin inserarea de spații între cuvinte. Uneori, pentru tabele, acest lucru nu este dorit și se invalidează cu comanda .nf (no fill). Revalidarea se face cu comanda .fi (fill).

Centrarea unor linii la mijlocul paginii se obține cu comanda:

.ce n

unde n este numărul următoarelor linii ce vor fi centrate.

Startarea unei noi pagini, prin abandonarea celei curente, se face cu comanda:

.bp

Deplasarea paginii cu un număr de caractere spre dreapta se obține cu comanda:

.po n

unde n este numărul de caractere (offset) cu care se face deplasarea.

Dacă se dorește mai mult spațiu între linii se utilizează comanda:

.ls n

unde n este numărul de spații dorit.

Dacă se dorește un singur spațiu de n linii se utilizează comanda:

.sp n

unde n este numărul de linii libere (lărgimea spațiului).

Identarea sau deplasarea unei linii spre dreapta se realizează cu comanda:

.ti n

unde n este numărul de caractere cu care se face deplasarea liniei.

Sublinierea a n linii se realizează cu comanda:

.ul n

unde n este numărul de linii de subliniat.

Utilizatorul are posibilitatea să-și definească noi comenzi prin intermediul așa-numitelor macro-instrucțiuni. O macro-instrucțiune se definește prin comanda:

.de nume-macroinstrucțiune

urmată de corpul macro-instrucțiunii și de comanda de sfîrșit macro-instrucțiune:

..

De exemplu, utilizatorul dorește o comandă care să-i lase 2 linii spațiu libere iar următoarea linie să fie deplasată cu 4 caractere spre dreapta. Macroinstrucțiunea este următoarea:

```
.de mi
.sp 2
.ti 4
..
```

Macroinstrucțiunile au avantajul că dau posibilitatea să se schimbe foarte repede formatul textului, fără a modifica comenzile **nroff**, ci numai macroinstrucțiunile utilizator.

Macroinstrucțiunea poate avea argumente, notate \$1, \$2, ..., \$9. De exemplu, macroinstrucțiunea de mai sus se poate defini cu argumente astfel:

```
.de mi
.sp \$1
.ti \$2
..
```

Ea se poate apela astfel:

.mi 2 4

Există și alte comenzi **nroff** (în total aproape 70 de comenzi) pentru crearea de antete de pagină, note de subsol, numerotare pagini etc. Există, de asemenea, o bibliotecă de macroinstrucțiuni **nroff** pentru elaborarea de rapoarte tehnice, articole științifice, literatură etc.

17.2.4.4. *Alte programe de formatare*

Programele de formatare **troff, neqn** și **tbl** intră în aceeași categorie cu **nroff.** Popularitatea sistemului UNIX rezultă și din existența acestor programe de formatare texte, care dau posibilitatea oricărui utilizator să-și elaboreze articole, manuale și altă documentație, rapid și eficient, într-un format identic cu cel realizat de tipografi.

Programul **troff** este asemănător cu **nroff,** dar cuprinde noi comenzi pentru a exploata facilitățile imprimantelor fotografice: caractere italice, dimensiuni diferite pentru litere, caractere îngroșate etc.

Programul **neqn** este utilizat pentru redactarea textelor cu formule matematice, chimice etc. Spre deosebire de **troff** el acceptă comenzi de descriere a ecuațiilor. De exemplu suma:

$$x = 2\,\Pi \sum_{i=1}^{n} a_i$$

se va descrie pentru **neqn** astfel:

x = 2 **pi sum from** i = 1 **to** n a **sub** i

Programul **neqn** recunoaște cuvintele cheie **pi, sum, from, sub** și transformă descrierea comenzii în comenzi efective pentru **troff**, în care intervin simbolii Π, Σ etc. Deci programul **neqn** apare ca un preprocesor pentru **troff.**

Redactarea tabelelor se realizează cu programul **tbl.** El este, de asemenea, un preprocesor pentru **troff** (sau **nroff**). Posedă comenzi speciale pentru delimitarea elementelor tabelelor, aranjarea pe linii, coloane, introducerea de caractere speciale, includerea în pătrățele etc.

Programul **spell** verifică despărțirea în silabe a cuvintelor din text, utilizînd un dicționar de cuvinte despărțite în silabe. De asemenea, se tipăresc cuvintele care nu există în direcționar, detectîndu-se astfel erori de genul 'pagnă' în loc de 'pagină'. Trebuie subliniat că nu se detectează erori de genul 'text' în loc de 'test'.

17.3. Mediul de programare în limbajul PASCAL

Însemnate preocupări atît pe plan național cît și pe plan mondial au ca obiectiv crearea unor medii de programare concentrate în jurul unor limbaje de programare cum ar fi ADA, PASCAL, LISP etc.

În acest paragraf vom prezenta în adapare după [118] mediul construit pentru dezvoltarea de programe în limbajul PASCAL. Acest mediu de programare își propune: să asiste programatorul pe parcursul întregului ciclu de viață al unui produs program; să constituie un suport pentru utilizarea limbajului PASCAL; să asigure gestiunea proiectului de elaborare a produsului program; să faciliteze programarea în echipă; să asigure justabilitate la nivelul instrumentelor; să fie extensibil în sensul de modificare și/sau extindere a mulțimii de instrumente pe care le înglobează.

În principal pentru fiecare etapă a procesului de elaborare a programelor este necesar să se asigure informații complete, corecte și precise privind starea curentă a proiectului precum și instrumente care să permită desfășurarea eficientă a activităților specifice etapei respective.

În acest scop baza de date a mediului de programare trebuie să înglobeze toate informațiile relevante asupra proiectului cum ar fi spre exemplu codul sursă, codul obiect, documentație etc.

Pentru atingerea obiectivelor enunțate mediului de programare PASCAL este structurat pe două niveluri și anume: un set de instrumente minim care este stabil; un set de instrumente extins care este variabil.

Primul nivel este de portabilitate al programatorului. Cel de-al doilea nivel include instrumente specifice diferiților membri ai echipei de programare.

În baza de date sînt memorate toate informațiile relevante pentru un proiect pe durata existenței sale. Fiecărui obiect al bazei de date i se asociază un corp (în format liber) și o mulțime de atribute.

Pe un set de obiecte se suprapune o varietate de rețele sau grafuri care reflectă relațiile între obiecte. În exploatarea acestei baze de date sînt importante noțiunile de *versiunile* și *derivatele* unui proiect.

Orice modificare a unui obiect produce o nouă versiune a acestuia. Derivatele înregistrează atît modul în care a fost creat un obiect, cît și informațiile folosite pentru producerea sa.

Instrumentele, pe cele două niveluri, operează asupra bazei de date creind și actualizînd obiecte.

Pentru nivelul unu al instrumentelor minime, se propune:

— *interpretorul de comenzi* folosit pentru a asigura comunicarea directă programator și mediul de programare;

— *editorul sintactic* cu ajutorul căruia programele sursă PASCAL sînt introduse în baza de date (acest editor înglobează componentele de analiză lexicală, sintactică și semantică a unui compilator);

— *generatorul de cod* care analizează forma intermediară construită de editorul sintactic și generează codul obiect (entitate în baza de date), propriu calculatorului pentru care se dezvoltă aplicația;

— *depanatorul simbolic* care pe baza execuției interpretative a formei intermediare asigură facilități de depanare simbolică interactivă cu culegerea și stocarea în baza de date de informații cu caracter statistic asupra execuției programului;

— *instrumente de evaluare a execuției* programului care pe baza datelor culese în faza de depanare generează o serie de informații referitoare la frecvența de execuție a structurilor de control ale programelor etc.

Un astfel de mediu de programare pentru limbajul PASCAL este în curs de realizare pentru minicalculatoarele românești INDEPENDENT și CORAL.

17.4. Instrumente software de dezvoltare a programelor pe mini- și microcalculatoare de producție românească

Fără de a avea intenția unei definiții, să precizăm că prin dezvoltarea de programe, în sens restrîns, înțelegem ansamblul activităților desfășurate din momentul scrierii programului pe un formular folosind un limbaj de programare și pînă la obținerea programului testat și documentat.

Așa cum am mai menționat, tendința este de a pune la dispoziția programatorilor un set de instrumente software, capabile să asiste programatorul pe parcursul dezvoltării programelor.

Corespunzător acestei tendințe, minicalculatoarele și microcalculatoarele de producție românească folosesc sisteme de operare moderne la care se adaugă o gamă diversificată de programe utilitare de ajutor în dezvoltarea de programe.

În cele ce urmează, pe structura celor două tipuri de sisteme de calcul, vom preciza posibilitățile oferite pe linia dezvoltării în condițiile de eficiență a programelor.

17.4.1. Sistem de operare și programe utilitare pe minicalculatoare INDEPENDENT și CORAL

Sistemul MIX, ca sistem actual de operare pe minicalculatoarele de producție românească INDEPENDENT și CORAL, este compatibil cu sistemul RSX-11M V4.1 al firmei DEC și permite programatorilor dezvol-

tarea programelor folosind un set bogat de instrumente software (programe utilitare).

Cadrul general de dezvoltare a programelor pe minicalculatoare este asigurat de următoarele instrumente software:

a) — *editoare de text* pentru crearea fișierelor cu programele sursă (exemplu: EDI și EDT);

b) — *compilatoare* pentru compilatoare de programare scrise în limbajele COBOL, FORTRAN, CPL/1, RTL 2, PASCAL, LISP, BASIC etc.;

c) — *editorul de legături* pentru asamblarea de module separate sau legarea la biblioteci și zone comune;

d) — *utilitare de lucru cu fișiere* cum ar fi spre exemplu PIP — pentru copiere, ștergere, modificare caracteristici fișiere; FLX — pentru conversii fișiere dintr-un sistem de operare în altul (DOS, RT, MIX, RSX); DMP — pentru listarea în diferite formate a conținutului unui fișier; CMP — pentru compararea fișierelor; LBR pentru crearea de biblioteci sursă sau obiect;

e) — *utilitare de lucru cu volume magnetice* dintre care enunțăm: BRU — pentru salvarea și restaura de date existente pe volume magnetice; DSC — pentru salvare și restaurare discuri magnetice; VFY — pentru verificarea consistenței informației sau datelor pe un disc magnetic; FMT — pentru formatarea și inițializarea discurilor magnetice; BAD — pentru detectarea blocurilor defecte pe discuri magnetice;

f) — *utilitare pentru depanarea și întreținerea programelor* cum ar fi spre exemplu: SLP — pentru corectarea fișierelor sursă; PAT — pentru corectarea fișierelor cu programele în format obiect; ZAP — pentru corectarea programelor în format executabil; ODT — pentru depanarea interactivă a programelor;

În principiu, dezvoltarea programelor scrise într-un limbaj evoluat se realizează parcurgînd etapele [110]:

A — *Crearea programului sursă.* Programele sistemului MIX, în marea lor majoritate, folosesc discul ca suport pentru memorarea fișierelor de intrare și respectiv de ieșire. În consecință și programele sursă trebuie mai întîi memorate pe disc. Operația de memorare a programelor sursă se poate realiza pe două căi și anume: folosind un editor de texte, cum ar fi spre exemplu EDI, pornind de la textul sursă introdus de la terminal sau prin copiere de pe un alt suport printr-un program de copiere gen FLX, PIP etc. (această din urmă variantă se folosește pentru copierea programelor de pe cartele);

B — *Compilarea.* În această etapă se obțin două fișiere de ieșire; opțiunile de compilare se descriu de către programator folosind ceea ce în limbaj uzual numim comutatoare;

C — *Crearea programelor executabile.* În această etapă sînt editate legăturile și stabilite toate caracteristicile programului executabil în funcție de cerințele precizate de către programator.

D — *Depanarea.* Activitatea de depanare a programelor se realizează în întreaga activitate de dezvoltare a programelor și se manifestă prin reveniri repetate, ceea ce impune posibilități de depanare în orice format al programului.

Să precizăm totuși că orice corecție trebuie efectuată, în mod obligatoriu și în cadrul formatului sursă. Pentru acest motiv prezintă interes deosebit instrumentele software pentru efectuarea de corecții la nivel format sursă cum ar fi spre exemplu EDI.

Așa cum am prezentat, pentru efectuarea de corecții în format obiect se poate folosi programul utilitar PAT, iar în format executabil programul utilitar ZAP.

Prin opțiunile de compilare sistemele implementate pe minicalculatoare ajută programatorul în depistarea erorilor și depanarea programelor.

Dezvoltarea de programe este susținută de o importantă bibliotecă obiect sistem. Procesoarele de limbaje conțin în principiu pe lîngă compilator și o bibliotecă de module obiect tipice. Pentru gestionarea bibliotecii se folosește programul bibliotecă LBR.

17.4.2. Dezvoltarea programelor în COBOL

Memorarea programelor COBOL în fișiere sursă pe suport magnetic (disc) se realizează conform celor două variante prezentate și anume folosind unul din programele utilitare EDI sau PIP.

Compilatorul COBOL [111] traduce programul sursă în module obiect relocabile (translatabile).

În urma compilării rezultă următoarele trei fișiere: fișierul obiect; (OBJ); fișierul de listare (LST); fișierul de reacoperire (SKL).

Fișierul obiect conține o mulțime de secțiuni (PSECT-uri) generate de compilator. Aceste secțiuni alcătuiesc un modul obiect care va fi supus unei editări de legături, eventual împreună cu alte module obiect pentru a se obține un program executabil. Prin urmare, se pot compila separat subprograme COBOL și apoi realiza legăturile între acestea.

Fișierul de listare conține o listă a textului sursă COBOL, erorile semnalate de compilator precum și alte liste opționale (tabele de alocarea memoriei, lista referințelor încrucișate etc.).

Fișierul de reacoperire conține directive care descriu structura de reacoperire a modulului obiect generat, din programul sursă COBOL. În acest fișier sînt generate directive pentru fiecare secțiune program reacoperibilă din textul sursă.

În urma compilării unei unități de program rezultă un modul obiect și un fișier de reacoperire pentru modulul respectiv.

Pentru utilizarea informațiilor din fișierele SKL, acestea trebuie reunite într-un fișier unic de reacoperire (ODL) care este acceptat de către editorul de legături.

Această operație se poate realiza folosind programul utilitar CBLMRG.

Programul utilitar CBLMRG are mai multe opțiuni prin care se pot alege caracteristici pentru fișierul de reacoperire ODL care să răspundă cel mai bine necesităților particulare ale utilizatorului.

Spre exemplu:

— se poate alege un fișier ODL „reunit" prin concatenarea tuturor fișierelor SKL de intrare după ce se adaugă instrucțiunile ODL generate de CBLMRG;

— se poate cere includerea depanatorului interactiv COBOL (CID) în task-ul executabil;

— se poate cere reacoperirea rutinelor de intrare/ieșire pentru a reduce dimensiunea task-ului executabil sau pentru a preveni depășirea spațiului de adrese al memoriei.

Utilizatorul poate crea sau modifica fișierul ODL și prin alte mijloace, cum ar fi spre exemplu folosind programul utilitar editor de texte (necesar îndeosebi cînd subprogramele sînt scrise în alt limbaj).

Editarea de legături cu sau fără un fișier ODL, în acest din urmă caz urmînd să se specifice direct în linia de comandă toate modulele obiect care trebuie legate. Fișierul obținut în ieșire conține programul executabil (task-ul).

Dintre instrumentele software specifice dezvoltării de programe COBOL, precizăm depanatorul interactiv COBOL simbolizat CID. Acest program poate fi inclus în programul COBOL în timpul editării de legături sau cu ajutorul utilitarului CBLMRG.

Cu ajutorul CID-ului se pot efectua următoarele operații: examinarea și modificarea datelor în programul COBOL; fixarea, afișarea și anularea punctelor de întrerupere; urmărirea execuției programului.

De precizat că CID-ul permite depistarea erorilor fără a modifica și recompila programul.

Pentru creșterea eficienței în dezvoltarea programelor COBOL se pot folosi biblioteci sursă și instrucțiunea COPY care permite includerea unui text sursă dintr-o bibliotecă eliminînd astfel codificarea repetată și erorile ce pot fi astfel generate. Textul sursă este tratat de către compilatorul COBOL ca și cum ar face parte din programul sursă respectiv (textul apelat prin copy trebuie să aibă același format).

Sistemul MIX admite două formate pentru introducerea unui program COBOL și anume:

— *formatul convențional*, numit și format cartelă este specific cazului cînd datele din liniile sursă sînt perforate pe cartele;

— *formatul terminal*, acceptat implicit de compilator în care o linie sursă are lungime maximă de 66 poziții, prima poziție servind pentru a marca comentările sau continuările de literali.

Programul utilitar CONFRM asigură trecerea programelor dintr-un format în altul.

Să reținem că în faza de editare a legăturilor se pot folosi elemente din biblioteca subprogramelor standard COBOL (ØTS) conținînd rutine apelate din modulele obiect generate de compilatorul COBOL și biblioteca subprogramelor standard (RMS) de intrare/ieșire ce pot fi apelate din modulele obiect generate de compilatorul COBOL.

17.4.3. Dezvoltarea de programe în FORTRAN

Pentru scrierea de programe FORTRAN sub controlul sistemului de operare MIX există trei compilatoare:

— *FORTRAN IV*, ce reprezintă standardul ISO din 1966 cu unele completări;

— *FORTRAN IV PLUS*, care prezintă unele extensii față de FORTRAN IV și generează cod pentru unitatea de virgulă flotantă (programele care lucrează cu variabile reale sînt mult mai rapide);

— *FORTRAN 77* ce reprezintă standardul limbajului FORTRAN din 1977 și prezintă o extensie a limbajului FORTRAN IV PLUS permițînd lucru cu secțiuni virtuale de date și lucru cu fișiere secvențial-indexate multicheie (el cere existența unității de virgulă flotantă). Se observă o abandonare a FORTRAN-ului IV și IV PLUS în favoarea FORTRAN-ului 77.

Crearea programelor sursă se face interactiv cu ajutorul unui editor de texte sau prin citire de pe cartele. Actualizarea și eliminarea erorilor se face, de asemenea, cu ajutorul editorului de texte sau prin programul SLP de corectare.

Editorul de legături este comun pentru toate limbajele de programare și realizează punerea în comun a mai multor module (subrutine) FORTRAN sau alte limbaje, la rezolvarea de referințe la subrutine din bibliotecile sistem

(SYSLIB, F4PLUS, F77OTS), legarea la biblioteci rezidente utilizator și la blocuri comune.

În ceea ce privește depanarea, eroarea este pusă în evidență la execuția programului, specificîndu-se numărul liniei de program în care a apărut și un mesaj explicativ. Utilizatorul urmează să consulte listingul programului FORTRAN, să verifice logica algoritmului, să corecteze programul cu editorul de texte, să-l recompileze să facă o nouă editare de legături și să-l execute.

Un alt instrument software pentru depanarea programelor FORTRAN îl constituie depanatorul simbolic interactiv. Cu acesta se poate face o execuție pas cu pas, se pot pune puncte de întrerupere în zonele cheie ale programului; se pot vizualiza variabile, tabele în timpul execuției după care se continuă execuția. La data elaborării prezentei lucrări este în curs de implementare un sistem de operare pe minicalculatoare compatibil cu sistemul UNIX.

17.4.4. Dezvoltarea programelor pe microcalculatoare

Microcalculatoarele românești FELIX M 18, FELIX M 18 B, FELIX M 118 etc. lucrează sub controlul sistemului de operare CP/M și permit dezvoltarea de programe folosind limbajele evoluate COBOL, FORTRAN, PASCAL, C, BASIC.

La acestea se adaugă limbajele specializate, cum ar fi spre exemplu dBASE II pentru gestionarea bazelor de date după modele ale bazelor de date relaționale.

Instrumentele software pentru dezvoltarea de programe sînt de genul celor prezentate pentru minicalculatoare.

Ca titlu de exemplu pentru dezvoltarea de programe în limbajul COBOL, se folosesc următoarele instrumente software: programul editor de texte ED; programul editor de texte WS*); programul compilator pentru programe scrise în limbajul COBOL **); programul de creare, listare, înregistrare, copiere și concatenare de fișiere pe disc flexibil PIP; programul CCP care asigură interfața simbolică între utilizator și sistemul CP/M; program de editare a legăturilor L80; module obiect pentru lucru din programe COBOL (cu terminale video în regim paginat); programul DDT de depanare interactivă a programelor obiect; programul SUBMIT care asigură lucrul în mod „batch" cu sistemul CP/M permițînd preluarea comenzilor CP/M nu de la consolă ci dintr-un fișier existent pe disc; programe de sortare ***).

În mod asemănător se poate vorbi de instrumente de dezvoltare de programe și aplicații în limbajul FORTRAN precum și în alte limbaje de programare.

---

*) Unul din cele mai evoluate editoare de texte la ora actuală.

**) Care permite lucrul cu fișiere organizate *secvențial-liniar*; *secvențial, indexat* sau *relativ* în condiții de acces *secvențial aleator* sau *dinamic* (cu respectarea relațiilor între organizare și acces).

***) Limbajul COBOL utilizat pe aceste calculatoare nu are implementat SORT-ul COBOL ori se știe că în dezvoltarea unor aplicații economice se întîlnesc frecvent operații de sortare.

17.5. Sisteme integrate de conducere şi elaborare a programelor mari: SICEP

Producerea pe principii industriale a programelor a devenit o necesitate tot mai resimţită în programare. În consecinţă eforturile au fost îndreptate spre dezvoltarea de sisteme de asistenţă în producerea îndeosebi a programelor mari.

Punerea în lucru şi integrarea metodelor, instrumentelor şi normelor în scopul de a face *P*rogramarea *A*sistată de *C*alculator s-au concretizat prin apariţia a ceea ce numim mediu de programare, sistem suport de dezvoltare programe, sistem de instrumente integrate, sisteme integrate de producere a programelor, atelier de programare etc.

Deşi denumirea şi semnificaţia acordată este diferită, beneficiile sînt aceleaşi şi constau în: creşterea calităţii; diminuarea costurilor; diminuarea timpului afectat elaborării programelor; ameliorarea condiţiilor de lucru în programare.

Cum pe plan naţional Sistemele Integrate de Conducere şi Elaborare a Programelor mari — SICEP — sînt, multe din ele, în faza de proiect, considerăm utile unele consideraţii teoretice şi practice, care să stea la baza conceperii unor astfel de sisteme, adaptate după [119—121].

17.5.1. Obiectivele unui SICEP

Un SICEP îşi fixează ca principale obiective:

A. *Ameliorarea productivităţii programelor.* Se ştie că productivitatea în programare este scăzută, ea se măsoară în cîteva instrucţiuni utile pe zi şi are ca principală cauză numeroasele iteraţii necesare pentru corectarea erorilor în diverse etape ale procesului de elaborare a unui program. Un factor important al calităţii rezidă în calitatea condiţiilor de lucru şi îndeosebi a celor care definesc interacţiunea utilizator-sistem, calităţile fizice şi funcţionale ale acestei interfeţe. Comunicarea internă, producerea documentaţiilor nu sînt mai puţin importante.

Sistemele pot lua în sarcină o mare parte din activităţile de rutină ale procesului de programare.

B. *Ameliorarea calităţii programelor obţinute.* Calitatea unui program este dată de modul cum sînt satisfăcute specificaţiile de funcţionare şi de performanţe.

Aspectele legate de facilităţile de exploatare şi de menţinere capătă o importanţă deosebită.

Intervin aspecte cum ar fi: capacitatea de evoluţie a programului pentru a răspunde la modificări de condiţii de exploatare sau materiale, facilităţile de iniţiere a utilizatorilor şi rapiditatea formării personalului. Asistenţa în documentare trebuie integrată în sistemul de dezvoltare.

C. *Creşterea utilizării efective a programelor.* Pentru ca un program să aibă largă utilizare trebuie să asigure realizarea cît mai uşoară a interfeţei om-calculator. Capătă o tot mai mare extindere folosirea tehnicii meniurilor în asigurarea acestei interfeţe. Facilitatea de a utiliza programul trebuie să devină o sarcină încă din faza de concepţie. Acest obiectiv a generat însemnate preocupări de studiu pe linia ergonomiei software-ului.

D. *Acoperirea tuturor etapelor ciclului de viață al programelor.* Așa cum s-a putut deduce din paragrafele precedente, multe din mediile de programare actuale au ca sferă de cuprindere etapele de codificare și eventual întocmirea documentației. Sistemele integrate de conducere și elaborare a programelor trebuie însă să acopere toate etapele ciclului de viață, începînd dacă se poate de la întocmirea specificațiilor și sfîrșind cu menținerea.

E. *Asigurarea interactivității.* Un astfel de sistem trebuie să asigure interactivitatea om-calculator în întreg procesul de elaborare a programelor. Postul de lucru este indicat să fie un terminal „inteligent".

F. *Integrarea într-un tot unitar a metodelor, instrumentelor, tehnicilor* capabile să asigure conducerea și elaborarea programelor asistată de calculator.

Fără a fi în intenția autorilor un studiu al obiectivelor să reținem că toate aceste obiective se subordonează obiectivului general și anume așezarea pe principii industriale și eficiență a întregii activități de programare.

17.5.2. Noțiuni, instrumente, concepte în organizarea SICEP

Analiza procesului de prelucrare a programelor conduce la definirea unei scheme conceptuale care precizează natura și caracteristicile diferitelor *obiecte* gestionate de către sistem; *relațiile* existente sau care trebuie să se stabilească și să se mențină între aceste obiecte; natura și efectul *operațiilor* (eroare, transformare, legătură etc.) care se pot realiza între aceste obiecte; compoziția acestor operații între ele.

Această schemă trebuie să se poată pune în lucru pentru o realizare fizică.

Forma generală a schemei conceptuale rezultă din obiectivele (ameliorarea productivității programatorilor și îmbunătățirea calității programelor obținute) și din caracteristicile de complexitate, evoluție și durată de viață.

Obiectele gestionate de către sistem vor fi *componente* ale programelor. Ca urmare a necesităților de punere la punct și de evoluție, aceste componente vor exista în *versiuni* multiple a căror coerență trebuie asigurată.

Noțiunea de componentă nu se limitează la texte de program, preocuparea de a integra ansamblul de instrumente conduce la considerația că un compus particular se manifestă în mai multe *forme* distincte (interfață, specificații, text sursă, program executabil, documentație etc.), adaptate fiecare la o tratare particulară și a căror coerență mutuală trebuie, de asemenea, menținută. Schema conceptuală a unui sistem integrat de conducere și elaborare a programelor se organizează în jurul unei *baze de date* în care sînt memorate componentele, sub forme diverse și în multiple versiuni. Diversele instrumente lucrează folosind această bază de date a cărei organizare trebuie să traducă constrîngerile fixate de către conceptorii produselor și cele impuse prin coerența sa internă.

Instrumentele software folosite în cadrul SICEP sînt programe care permit să se efectueze sau să se îndeplinească o activitate. Ele ușurează o activitate manuală sau intelectuală și se clasifică în:

— *Instrumente conceptuale* care incorporează metode de programare, de testare, de descompunere ierarhică, de ascundere a informațiilor etc. Ele sînt orientate către tehnici de rezolvare de probleme utilizate în dezvoltarea sistemelor.

— *Instrumente tehnice,* sînt instrumente de producție, de control și de gestiune care ajută în procesul dezvoltării programelor, validării și planificării lor.

— *Instrumente lingvistice*, sînt limbaje și oferă mijloace de exprimare și de comunicare a ideilor și permit formularea conceptelor sau problemelor de rezolvat (vezi și anexa).

Actualele medii de programare oferă instrumente cu precădere sau în exclusivitate din ultimele două categorii.

Instrumentele, îndeosebi cele tehnice, se mai disting în funcție de etapele ciclului de viață (specificație, concepere, codificare, asigurare calitate, validare) sau funcționalitate (dezvoltare, gestiune proiect, măsurarea performanțelor).

Un instrument software este caracterizat prin fluxul de informații care constituie intrările și ieșirile văzute de către utilizator.

Un concept deosebit în elaborarea programelor mari este *modularitatea*.

Așa cum am prezentat și în volumul 1, modularitatea se traduce prin:

— descompunerea sistemului în *module*, fiecare fiind definit printr-o specificație funcțională care descrie în întregime comportarea sa fără a face să intervină detaliile de realizare;

— posibilitatea de a descrie un sistem de manieră ierarhică separînd pe diferite niveluri structura sa globală (asamblarea părților) de structura sa de detaliu (descrierea fiecărei părți).

Un modul este asocierea unei *interfețe* care descrie în întregime resursele furnizate la utilizatorii * săi și un *corp* care realizează concret resursele.

Interfața și corpul utilizează la nevoie resursele furnizate de alte module. În practică aceste resurse sînt constante, variabile, proceduri.

Specificația externă a unui modul descrie comportamentul său pentru un utilizator făcînd să intervină numai resursele vizibile furnizate prin interfață. *Specificația de realizare* a corpului modulului descrie structura sa internă; ea este inaccesibilă utilizatorilor modulului. Trei tipuri de relații apar între diferite obiecte și anume:

— *relația de descriere* între o specificație și obiectul la care ea se aplică;

— *relația de ralizare* între interfața unui modul și corpul care realizează resursele pe care ea le furnizează;

— *relația de utilizare* între un modul A și un modul B a cărui realizare pune în lucru resursele furnizate de către A.

Interesul cu privire la aceste relații vine din faptul că ele exprimă constrîngeri de coerență care trebuie menținute cînd un obiect este modificat sau înlocuit. Un aspect important al acestor relații este deci posibilitatea verificării automate de către SICEP. Exploatarea relațiilor de descriere ține de domeniul instrumentelor software de ajutor în conceperea programelor.

Într-un SICEP ne putem mulțumi cu detectarea și semnalarea oricărei modificări care afectează unul din termenii descrierii.

Relația de realizare este în general exprimată incomplet, în limbajele existente, printr-o simplă identitate textuală: lista resurselor furnizate de către interfață este reprodusă în corp, cu numele și tipul lor.

Relația de utilizare este un caz particular al *relației de dependență*. Se spune despre un sistem *A* că depinde de un sistem *B* dacă buna funcționare a lui *A* necesită prezența lui *B*. Din examinarea programelor se poate deduce dependența.

Relația de dependență impune un ordin parțial pentru compilare și permite să se determine consecințele unei modificări asupra unui compus al unui sistem.

---

* Un om sau un alt modul.

Modularizarea face ca prin compoziție (operație care permite să se construiască un sistem complex pornind de la componentele sale) să se obțină programe complexe.

Se disting două aspecte și anume: exprimarea compoziției și construirea propriu zisă.

În sistemele clasice exprimarea compoziției se limitează la lista de comenzi date editorului de legături prin care se specifică fișierele care conțin componentele de legat între ele.

La sistemele de dată recentă se disting două forme de exprimare a compoziției:

— *Expresia globală* folosind un limbaj care permite să se definească obiectele de genul modulelor sau interfețelor și operații cum ar fi legătura. Un aspect esențial este verificarea compatibilității tipurilor de resurse de intrare/ieșire.

Limbajul de conexiune permite să se numească componentele, să stabilească relațiile de realizare între corp și interfață, să verifice compatibilitatea tipurilor de resurse de intrare/ieșire. Mai multe interfețe pot fi asociate aceluiași corp; iar mai multe corpuri pot coopera la realizarea unei interfețe.

— *Expresia relației de dependență.* Cînd fiecare compus conține lista compușilor de la care primește intrările sau pe care le include, expresia, compoziției sistemului este repartizată între diversele sale componente și poate fi reconstituită pornind de la modulele principale prin închidere tranzitorie a relației de dependență.

Cum *componentele* și *sistemele* există în versiuni multiple un aspect important vizează gestionarea acestora.

În anumite cazuri, crearea unei noi versiuni *) face perimată versiunea precedentă. Sînt însă și cazuri cînd versiunile coexistă.

Faptul că mai multe versiuni coexistă presupune însă determinarea versiunii fiecărui component pentru o anumită versiune a sistemului global pe care îl construim precum și exprimarea și punerea în lucru a regulilor de compatibilitate sau de implicare între versiunile diferitelor componente.

O problemă deosebit de importantă pentru conceperea unui SICEP o constituie cea a reprezentărilor interne și construirii unui component program. În schema tradițională a etapelor tratării unui program un component program parcurge o suită de transformări între *forma scrisă* și *forma executabilă.*

Se poate aprecia că: fiecare instrument consideră de o manieră diferită componenta tratată; orice modificare a textului sursă necesită trecerea prin toate etapele parcurse în procesul de transformare; componentele program sînt păstrate sub forme care permit reutilizarea lor.

Integrarea diverselor transformări într-un sistem de producție vizează reducerea lucrărilor inutile și facilitarea interacțiunii utilizator-program precum și ușurarea punerii în comun și reutilizarea componentelor.

Prin urmare se caută: să se aleagă pentru componente o formă internă care poate fi utilizată de către mai multe instrumente, rămînînd totuși ușor folosibilă de către programator; limitarea la strictul necesar a consecințelor unei modificări locale fără să impună retratarea părților nemodificate.

Prezintă interes reprezentări interne derivate din arborii sintaxici (utilizați pînă acum în interiorul compilatoarelor).

---

*) În principiu versiunile corespund la modificări care au semnificație logică.

Arborele abstract sintaxic nu reprezintă singur toate informațiile conținute în program. Nodurile sale trebuie să conțină atribute. Fiecare instrument care folosește această reprezentare internă utilizează un anumit subansamblu din aceste atribute.

Acțiunea unui instrument (de manipulare și vizualizare, de generare de cod sau de documentare) asupra unei astfel de forme de reprezentare poate, în numeroase cazuri, să fie interpretată ca o modificare *de atribute* și nu a *structurii* de bază.

17.5.3. Cadrul de producție

Un SICEP presupune în primul, rînd un cadru de producție. În literatura de specialitate acesta mai este numit și atelier de programare (sau atelier de programare integrat).

Fără a încerca o definiție (de altfel și greu de dat), să spunem că prin *atelier de programare înțelegem un cadru de producție în care programele nu mai sînt concepute de o manieră artizanală ci de o manieră industrială. Ele oferă un cadru de producere a programelor pe principii industriale bazat pe asistență din partea calculatorului.*

Se poate deci spune de o manieră generală „că un atelier de programare este gîndit ca un sistem organizat cu structurile sale, persoanele sale, sarcinile sale repartizate între persoane".

Apreciem deci un atelier de programare prin: mașinile care definesc mediul material și constituie arhitectura atelierului; relațiile om-mașină propuse; metodele aplicate pentru procesul elaborării programelor.

Vocația acestor ateliere de programare este de a asigura o abordare coerentă procesului de elaborare a programelor și de a permite astfel creșterea eficienței în programare și ușurarea muncii programatorilor.

Un aspect important al atelierelor de programare este integrarea, care constă în implementarea în cadrul unei configurații materiale adaptate a instrumentelor compatibile și destinate lucrului unitar, ușurînd aplicarea metodelor și folosirea limbajelor atelierului.

Aspectele vizate de integrare sînt: asistența din partea calculatorului pe toate etapele elaborării programelor; urmărirea elaborării programelor (planificare, evidență stadiu de avansare, controlul termenelor); interacțiunea om-atelier și luarea în calcul a considerentelor de ordin economic; asigurarea controlului de calitate.

Termenul însă de atelier de programare este generic. În general atelierele se adaptează tipului de aplicație vizat.

Una din primele preocupări ale oricărui atelier de programare a fost de a permite utilizatorilor de a aplica regulile fundamentale ale unui program corect, de a avea acces cu ușurință la resursele de prelucrare a programelor și de a folosi cu ușurință sistemele de operare.

Foarte repede ideea *postului de lucru interactiv* s-a dovedit indispensabilă pentru o comunicare convențională cu mașina. Descentralizarea, utilizarea terminalelor inteligente echipate cu instrumente indispensabile lucrului individual racordate la rețele locale; cooperarea între diferiți utilizatori (care presupune că anumite lucrări sînt planificate, urmărite, ghidate și că etapele sînt verificate pentru a fi coerente) constituie doar cîteva din coordonatele esențiale ale unui atelier modernizat de programare.

În [119] se consideră că în evoluția atelierelor de programare s-au conturat trei generații.

Prima generație se caracterizează prin folosirea unui calculator (de obicei minicalculator) pentru dezvoltarea de programe. Utilizatorul lucrează folosind un post interactiv care îi permite să aibă acces la un număr de instrumente axate în principal pe scrierea programelor, întocmirea documentațiilor și menținerea lor. De remarcat că în această primă generație utilizatorul caută să folosească cît mai bine instrumentele sistemului. Totodată se face distincție între calculatorul folosit în dezvoltarea de programe și cel folosit în exploatarea programelor.

Referitor la tehnica de programare se poate spune că atelierele din această generație prin instrumentele puse la dispoziția programatorilor nu depășesc cadrul clasic (compilatoare, asambloare, editoare de legături). Se fac puține abordări de ordin metodologic și în legătură cu gestionarea proiectelor.

A doua generație are la bază conceptul de *producție a programelor*. Utilizatorului i se propune un ansamblu de instrumente care îl vor ajuta pe parcursul ciclului de viață al programelor. Anumite metode pot fi asociate dezvoltării de programe și ghidări în concepție.

Se integrează astfel principiile ingineriei programării dezvoltînd instrumentele și metodele de concepție și producție asistată de calculator pentru a garanta calitatea și productivitatea programelor.

Totodată se înregistrează trecerea de la artizanat la producția industrială și aceasta îndeosebi folosind sisteme de gestiune și dezvoltarea programelor (SGDP) constituite în acest scop. Accentul este pus pe mijloacele informatice care oferă un mediu de programare dotat de un ansamblu complet de instrumente. Sistemul UNIX a influențat substanțial sistemele de gestiune și dezvoltare de programe.

A treia generație concretizează atelierele de programare (aflate multe dintre ele în faza de proiect) care reunește ideile cele mai noi în domeniul ingineriei programării orientate către integrarea instrumentelor și metodelor. Această generație își fixează ca obiectiv să ia în calcul fazele de la concepere la menținere (unele chiar și faza de întocmire a specificațiilor). Precizăm faptul că aceste ateliere sînt dotate cu sisteme de gestiune a bazei de date a programelor produse cu mecanisme de control a coerenței. Sînt studiate totodată probleme de coerență a atelierelor de programe. Mijloacele citate sînt reprezentările interne, unicitatea limbajului și a metodei, unicitatea cîmpului de aplicație și utilizarea aceloraşi meta-instrumente.

Conceptele cheie ale acestei de a treia generații de ateliere sînt: descentralizarea, integrarea și modularizarea.

Funcțiile îndeplinite de un atelier de programare din această generație sînt: funcția de producție; funcția de comunicație; funcția de control-calitate; funcția de arhivare/protecție; funcția de conducere a proiectului.

Ca tendință se apreciază două căi de urmat pentru concepție de ateliere:

— una care propune o structură care să permită ajutor acordat utilizatorului pentru a construi propriile sale instrumente pornind de la instrumente parametrabile sau meta-instrumente; este calea meta-atelierelor de producere a programelor lăsînd liber curs pentru adaptare și evoluție;

— una care promovează independența față de o metodă, de un limbaj, sau de o reprezentare internă care se justifică prin posibilitatea de a dispune

de instrumente mai eficace și mai bine armonizate datorită cunoașterii precise a mecanismelor de construire a programelor.

Dacă concepția unui atelier de programare rămîne o problemă de specialist, utilizarea sa trebuie să se deschidă tuturor: conceptori, analiști, programatori, șefi de proiecte etc. Datorită acestor ateliere informațiile vor circula mai bine, va crește capacitatatea de lucru a echipelor de programare iar coordonarea lucrării devine posibilă.

Iată de ce atelierele de programare integrate sînt acum indispensabile proiectării de programe mari.

A nu se uita totuși faptul că producția de programe este o activitate umană.

Competențele de care depinde în cele din urmă succesul unui proiect nu sînt automatizabile. Instrumentele perfecționate puse la dispoziție de către tehnologie nu se pot substitui *omului*, ele îl ajută însă să-și exprime mai bine talentul și creativitatea.

18. Programarea folosind limbaje de nivel înalt

18.1. Introducere

Dezvoltarea unor medii de programare puternice, mai întîi pe sistemele de calcul actuale iar ulterior şi pe calculatoarele personale, pun bazele unor veritabile sisteme personale pentru dezvoltarea de software. În cadrul acestor sisteme o pondere tot mai mare o vor avea limbajele de nivel înalt cu precădere neprocedurale sau descriptive. Aceste limbaje sînt gîndite cît mai apropiate de limbajele uzuale, iar prin caracterul lor neprocedural sau descriptiv permit utilizatorului ca la nevoie, să-şi dezvolte singur aplicaţii de mai mică complexitate. În acest capitol se descrie succint limbajul dBASE care permite lucru cu baze de date relaţionale şi care este implementat pe microcalculatoare sub controlul sistemului de operare CP/M.

Limbajul dBASE conţine comenzi care fac deschiderea spre ceea ce numim programarea neprocedurală, păstrînd intacte posibilităţile programării procedurale (vezi lucru cu fişiere de comenzi). Mai menţionăm faptul că limbajul dBASE poate fi utilizat şi în activitatea de proiectare ca instrument de verificare a proiectului. Literatura de specialitate menţionează şi alte limbaje mai performante decît dBASE, dar neimplementate încă pe calculatoarele de producţie românească. Este cazul limbajelor de lucru cu baze de date relaţionale cum ar fi: limbajul SEQUEL, limbajul QBE (QUERY BY EXAMPLE) şi limbajul QUEL.

Cel mai „înalt" pare a fi limbajul RANDEVOUS, definit de reputatul specialist E. F. Codd.

18.2. Un exemplu practic de programare folosind un limbaj de nivel înalt

Se presupune că un anume utilizator lucrează în domeniul evidenţei personalului şi a retribuţiilor şi dispune de un microcalculator care lucrează sub controlul sistemului de operare CP/M precum şi de procesorul numit dBASE care-i permite să gestioneze date după modelele bazelor de date relaţionale folosind limbajul de nivel înalt dBASE. Pentru a-şi uşura munca, utilizatorul decide să facă apel la calculator. Avînd cunoştinţe asupra limbajului dBASE decide să-şi dezvolte singur o aplicaţie care să-i permită realizarea lucrărilor de gestiune a personalului. Mai întîi decide să memoreze

datele cu privire la personalul întreprinderii (pentru simplificare nu le vom preciza pe toate) într-un fişier de date (cu extensie DBF) în accepţiunea limbajului dBASE.

În acest scop foloseşte comanda CREATE care-i permite să precizeze identificatorii, tipul datelor şi lungimea acestora. Spunem că în acest mod s-a definit structura datelor care constituie şi prima înregistrare a fişierului de date în accepţiunea limbajului dBASE.

În figura 18.1 *a* se prezintă listingul cu elementele dialogului adecvat comenzii CREATE pentru crearea fişierului intitulat PERSONAL (s-au subliniat elementele specificate în cadrul dialogului de către utilizator), iar în figura 18.1*b* structura înregistrării fişierului creat obţinută prin comandă „DISPALY STRUCTURE"

```
m18b 53k CP/M vers 2.2/1.1

A>DBASE

ENTER TODAYS DATE AS MM/DD/YY          OR RETURN FOR NONE  : 12/03/85

*** dBASE II      Ver 2.
.
. CREATE
ENTER FILENAME: PERSONAL
ENTER RECORD STRUCTURE AS FOLLOWS:
 FIELD    NAME,TYPE,WIDTH,DECIMAL PLACES
 001      LOC:MCA,N,4
 002      MARCA,N,5
 003      NUME,C,15
 004      PRENUME,C,15
 005      FUNCTIA,C,2
 006      RETRIB,N,4
 007      AN:NAST,N,6
 008      AN:ANGAJ,N,6
 009      STARE:CIV,C,1
 010      NR:COPII,N,1
 011      SEX,C,1
 012
INPUT DATA NOW?
```

Fig. 18.1 *a*. — Precizarea structurii datelor.

```
.
. USE PERSONAL
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:   PERSONAL.DBF
NUMBER OF RECORDS:    00010
DATE OF LAST UPDATE:  00/00/00
PRIMARY USE DATABASE
FLD        NAME        TYPE WIDTH   DEC
001      LOC:MCA        N    004
002      MARCA          N    005
003      NUME           C    015
004      PRENUME        C    015
005      FUNCTIA        C    002
006      RETRIB         N    004
007      AN:NAST        N    006
008      AN:ANGAJ       N    006
009      STARE:CIV      C    001
010      NR:COPII       N    001
011      SEX            C    001
** TOTAL **             00061
.
.
```

Fig. 18.1 *b*. — Structura articolului fişierului creat.

Prin afişarea numărului 012 în dreptul coloanei, sistemul invită utilizatorul să descrie un nou cîmp de date; acesta întrucît a descris toate cîmpurile ale căror valori vrea să le memoreze în fişier, acţionează tasta ⟨CR⟩ sau ⟨RETURN⟩ (în funcţie de tipnl terminalului). Ca răspuns, sistemul invită utilizatorul prin mesajul INPUT DATA NOW? să introducă datele.

Prin acţionarea tastei Y utilizatorul poate cere continuarea dialogului pentru introducerea datelor. Pe ecranul terminalului apar descrise cîmpurile şi se cere introducerea datelor.

De la terminal, utilizatorul introduce datele pentru fiecare din înregistrări. Dacă data de introdus ca valoare a unui cîmp are lungimea în caractere egală cu lungimea maximă a cîmpului, sistemul asigură poziționarea automată în dreptul cîmpului următor pentru introducerea datelor; în caz contrar este necesar să se acționeze tasta ⟨CR⟩ pentru a asigura alinierea datelor în cadrul cîmpului și poziționarea în dreptul cîmpului următor (sistemul asigură un set de taste de control pentru operare în mod ecran). După ce s-au introdus datele corespunzătoare ultimului cîmp din înregistrare, sistemul memorează înregistrarea în fișierul de date PERSONAL și afișează pe ecran din nou imaginea înregistrării pentru a se introduce datele în legătură cu o altă persoană.

Procesul de introducere a datelor și înregistrare în fișier se repetă pînă cînd utilizatorul acționează tasta ⟨CR⟩ la apariția pe ecran a imaginii înregistrării pentru a introduce noi date. Pointerul se va poziționa în dreptul primului cîmp din structura înregistrării.

Utilizatorul poate cere, pentru control, lista tuturor înregistrărilor din fișier dînd comenzile USE PERSONAL pentru activarea fișierului de personal și LIST (sau DISPLAY) pentru listarea tuturor înregistrărilor (fig. 18.2).

```
. USE PERSONAL
. LIST
00001  1011  4131 POPESCU        NICOLAE    EC  4000  130347  150270 C  2 M
00002  1011  4151 IONESCU        ION        EC  3800  120249  110372 C  1 M
00003  1011  4171 SLAVESCU       MARIA      IN  3300  110453  130178 C  1 F
00004  1011  4191 GEORGESCU      ION        FU  2500  130749  120772 C  1 M
00005  1011  1234 IONESCU        VASILE     EC  4560  120349  120569 C  1 M
00006  1022  4301 ALBESCU        VASILE     EC  2850  120558  110880 C  0 M
00007  1022  4501 DUMITRU        ION        IN  4450   10147   20270 N  0 M
00008  1033  4541 RADU           MIRCEA     EC  3500  121158  110280 C  0 M
00009  1033  4571 BIDASCU        CARMEN     EC  3700  120157  130480 C  1 F
00010  1234  1234 VASILESCU      EUGENIU    EC  5200  120338  120459 C  2 M
```

Fig. 18.2. — Exemplu de utilizare a comenzilor USE și LIST.

El poate cere lista selectivă a persoanelor cu funcția „ECONOMIST" pentru care cîmpul FUNCȚIA are valoarea „EC" (figura 18.3 *a*) sau a persoanelor cu funcția economist și retribuția tarifară > 3 000 (fig. 18.3 *b*).

```
. LIST FOR FUNCTIA = 'EC'
00001  1011  4131 POPESCU        NICOLAE    EC  4000  130347  150270 C  2 M
00002  1011  4151 IONESCU        ION        EC  3800  120249  110372 C  1 M
00005  1011  1234 IONESCU        VASILE     EC  4560  120349  120569 C  1 M
00006  1022  4301 ALBESCU        VASILE     EC  2850  120558  110880 C  0 M
00008  1033  4541 RADU           MIRCEA     EC  3500  121158  110280 C  0 M
00009  1033  4571 BIDASCU        CARMEN     EC  3700  120157  130480 C  1 F
00010  1234  1234 VASILESCU      EUGENIU    EC  5200  120338  120459 C  2 M
```

Fig. 18.3 *a*. — Comanda LIST cu opțiunea FOR.

```
. LIST FOR FUNCTIA = 'EC' .AND. RETRIB > 3500
00001  1011  4131 POPESCU        NICOLAE    EC  4000  130347  150270 C  2 M
00002  1011  4151 IONESCU        ION        EC  3800  120249  110372 C  1 M
00005  1011  1234 IONESCU        VASILE     EC  4560  120349  120569 C  1 M
00009  1033  4571 BIDASCU        CARMEN     EC  3700  120157  130480 C  1 F
00010  1234  1234 VASILESCU      EUGENIU    EC  5200  120338  120459 C  2 M
```

Fig. 18.3 *b*. — Comanda LIST cu opțiunea FOR și o condiție compusă.

Cum deseori în munca de evidență a personalului se cer informații numai despre anumite persoane, utilizatorul, pentru a facilita accesul la înregistrări poate proceda la indexarea fișierului de date în funcție de valorile unuia sau mai multor cîmpuri. Spre exemplu, el poate constitui un index pe valorile cîmpului NUME din structura înregistrărilor iar ulterior prin

comenzile FIND și DISPLAY poate cere informații numai despre o anume persoană (fig. 18.4).

Pentru a edita un raport cu personalul din evidența întreprinderii, utilizatorul descrie mai întîi structura acestuia folosind comanda REPORT în formatul REPORT FORM SIT1 (fig. 18.5*a*). Descrierea raportului este memorată într-un fișier distinct, creat de sistem, cu numele raportului și extinsia ⟨.FRM⟩, după modelul din figura 18.5*b*.

```
.
.
.
.
.
. USE PERSONAL
. INDEX ON NUME TO FINDEX.NDX
00008 RECORDS INDEXED
.
.
. FIND 'BIDASCU'
. DISPLAY
00008   1033   4571 BIDASCU        CARMEN        EC  3700  120157  130480 C  1 F
.
. FIND &VAR:SIR
. DISPLAY
00007   1033   4541 RADU           MIRCEA        EC  3500  121158  110280 C  0 M
.
.
.
```

Fig. 18.4. — Indexarea unui fișier și căutarea rapidă de înregistrări.

```
.
.
. REPORT FORM SIT1
ENTER OPTIONS, A=LEFT MARGIN, L=LINES/PAGE, W=PAGE WIDTH
PAGE HEADING? (Y/N) Y
ENTER PAGE HEADING: LISTA PERSONAL PE LOCURI DE MUNCA
DOUBLE SPACE REPORT? (Y/N)
ARE TOTALS REQUIRED? (Y/N) Y
SUBTOTALS IN REPORT? (Y/N) Y
ENTER SUBTOTALS FIELD: LOC:MCA
SUMMARY REPORT ONLY? (Y/N)
EJECT PAGE AFTER SUBTOTALS? (Y/N)
ENTER SUBTOTAL HEADING: LOC MUNCA
COL     WIDTH,CONTENTS
001     5,LOC:MCA  <CR>
ENTER HEADING: ====LOC;=====
ARE TOTALS REQUIRED? (Y/N)
002     15,NUME
ENTER HEADING: ==============NUME;======\========
003     15,PRENUME
ENTER HEADING: ==============PRENUME;==============
004     8,RETRIB
ENTER HEADING: ========RETRIB;=======
ARE TOTALS REQUIRED? (Y/N) Y
005
```

Fig. 18.5 *a*. — Descrierea unui Raport (modelul unei situații).

```
. TYPE SIT1.FRM
Y
LISTA PERSONAL PE LOCURI DE MUNCA

Y
Y
LOC:MCA

LOC MUNCA
5,LOC:MCA
====LOC;=====

15,NUME
==============NUME;======\========
15,PRENUME
==============PRENUME;==============
8,RETRIB
========RETRIB;=======
Y
```

Fig. 18.5 *b*. — Structură fișier FRM.

Datele din structura rîndului de detaliu sînt preluate din fișierul PERSONAL (cu extensia .DBF)

Folosind comanda REPORT în formatul (vezi și fig. 18.6)

REPORT FORM SIT1 TO PRINT

utilizatorul cere obținerea (listarea) raportului și la imprimanta (TO PRINT). Pentru construirea raportului se folosesc datele din fișierul activ, în cazul de față numit PERSONAL (dacă acesta nu este activ utilizatorul prin comanda USE PERSONAL asigură activarea).

Pentru ordonarea înregistrărilor în funcție de valorile crescătoare ale cîmpului LOC:MCA și editarea situației SIT1 se dau comenzile din figura 18.7. Efectul este cel din figura 18.8.

În cazul în care utilizatorul dorește să editeze și la imprimantă același raport numai pentru personalul cu studii economice care-și desfășoară activitatea pe fiecare loc de muncă dă comandă REPORT în formatul:

REPORT FORM SIT1 FOR FUNCȚIA = 'EC' TO PRINT.

Efectul este redat în figura 18.8*b*.

```
. USE PERSONAL
. REPORT FORM SIT1


PAGE NO. 00001

                    LISTA PERSONAL PE LOCURI DE MUNCA

===== ============== ============== ========
 LOC        NUME          PRENUME     RETRIB
===== ======\======== ============== --------

* LOC MUNCA   1011
 1011 POPESCU          NICOLAE          4000
 1011 IONESCU          ION              3300
 1011 SLAVESCU         MARIA            3300
 1011 GEORGESCU        ION              2500
** SUBTOTAL **                         14600


* LOC MUNCA   1022
 1022 ALBESCU          VASILE           2850
 1022 DUMITRU          ION              4450
** SUBTOTAL **                          7300


* LOC MUNCA   1033
 1033 RADU             MIRCEA           3500
 1033 BIDASCU          CARMEN           3700
** SUBTOTAL **                          7200


* LOC MUNCA   1011
 1011 IONESCU          VASILE           2390
** SUBTOTAL **                          2390


* LOC MUNCA   1234
 1234 VASILESCU        EUGENIU          5200
** SUBTOTAL **                          5200

** TOTAL **                           37860
```

Fig. 18.6. — Editarea unui raport (folosind în intrare un fișier nesortat).

```
. USE PERSONAL
. SORT ON LOC:MCA TO SPERS
SORT COMPLETE
. USE SPERS
. REPORT FORM SIT1 TO PRINT
```

Fig. 18.7. — Sortarea unui fișier.

Prin operațiunea FOR s-a specificat că se face o raportare selectivă. FUNCȚIA = 'EC' reprezintă condiția de verificat.

De o manieră asemănătoare, adică prin dialog utilizator-sistem, folosind comenzi cu o structură relativ simplă, utilizatorul modifică structura fișierului, adaugă noi înregistrări în fișier, anulează înregistrări, modifică valorile anumitor cîmpuri din toate înregistrările sau în mod selectiv, cere furnizarea unor informații la nivel fișier etc.

Desigur că pentru prelucrări mai complexe, pentru care trebuie date într-o anumită succesiune mai multe comenzi, este indicat ca ele să fie memo-

```
PAGE NO. 00001

                    LISTA PERSONAL PE LOCURI DE MUNCA

===== ============== ================ ========
 LOC        NUME          PRENUME       RETRIB
===== ======\========= ================ ========

* LOC MUNCA   1011
 1011 POPESCU          NICOLAE            4000
 1011 IONESCU          ION                3800
 1011 SLAVESCU         MARIA              3300
 1011 GEORGESCU        ION                2500
 1011 IONESCU          VASILE             4560
** SUBTOTAL **
                                         18160

* LOC MUNCA   1022
 1022 ALBESCU          VASILE             2850
 1022 DUMITRU          ION                4450
** SUBTOTAL **
                                          7300

* LOC MUNCA   1033
 1033 RADU             MIRCEA             3500
 1033 BIDASCU          CARMEN             3700
** SUBTOTAL **
                                          7200

* LOC MUNCA   1234
 1234 VASILESCU        EUGENIU            5200
** SUBTOTAL **
                                          5200

** TOTAL **
                                         37860
```

Fig. 18. 8 *a*

```
PORT FORM SIT1 FOR FUNCTIA = 'EC'

PAGE NO. 00001

                    LISTA PERSONAL PE LOCURI DE MUNCA

===== ============== ================ ========
 LOC        NUME          PRENUME       RETRIB
===== ======\========= ================ ========

* LOC MUNCA   1011
 1011 POPESCU          NICOLAE            4000
 1011 IONESCU          ION                3800
 1011 IONESCU          VASILE             4560
** SUBTOTAL **
                                         12360

* LOC MUNCA   1022
 1022 ALBESCU          VASILE             2850
** SUBTOTAL **
                                          2850

* LOC MUNCA   1033
 1033 RADU             MIRCEA             3500
 1033 BIDASCU          CARMEN             3700
** SUBTOTAL **
                                          7200

* LOC MUNCA   1234
 1234 VASILESCU        EUGENIU            5200
** SUBTOTAL **
                                          5200

** TOTAL **
                                         27610
```

Fig. 18.8 *b* — Editarea unui raport folosind în intrare un fişier sortat.

rate mai întîi într-un fișier distinct numit fișier de comenzi și lansate apoi în execuție.

Spre exemplu, dacă utilizatorul vrea să verifice datele înainte de a fi introduse în fișierul de date poate opta pentru organizarea unei proceduri cu funcție de verificare a corectitudinii datelor și să înregistreze numai datele corecte în fișierul PERSONAL *).

Pentru organizarea procedurii se pornește de la următoarele specificații: fișierul personal are structură prezentată în figura 18.1 *b*; datele se introduc de la terminal; se va da posibilitatea utilizatorului să facă corecții după introducerea datelor; controlul datelor vizează cu precădere mulțimea datelor posibile;

Pentru realizarea procedurii se poate consulta exemplul din figura 18.14.

Înregistrarea procedurii într-un fișier de comenzi cu extensia .CMD se realizează folosind un editor de texte care poate fi *ED sau WS*.

18.3. Elemente de bază ale limbajului dBASE

În procesul de elaborare de programe în limbajul dBASE se operează cu următoarele elemente de bază:

A. *Constante* care au valoare invariabilă în program și participă ca atare în prelucrare. Exemple: constantă șir: "INTR: DACIA" sau 'INTR: DACIA'; constantă numerică: 3 sau 743.21; constantă logică: .T. pentru adevărat,.F. pentru fals.

B. *Variabile* care pot fi de memorie (în număr de maxim 64 într-un program) sau cîmpuri din înregistrări (în număr de maxim 32 într-o înregistrare). Ca și constantele și variabilele pot fi de tip *numeric* (N), *șir de caractere* (C) sau *logic* (L). Un nume de cîmp se compune din maxim 10 caractere alfanumerice plus caracterul „:" (care poate să apară numai în interior) și trebuie să înceapă cu o literă. Exemple: A

NR:FACT

COPIL1

La descrierea unui cîmp se precizează numele, natura și dimensiunea. În cazul cîmpurilor numerice, la calcularea dimensiunii cîmpului trebuie inclusă și poziția pentru punctul zecimal.

C. *Fișiere* **) care pot fi:

1) *De date* (cu extensia .DBF) constituite dintr-o înregistrare de structură și 0 la 65 535 înregistrări de date. În momentul memorării înregistrărilor de date în fișier, fiecărei înregistrări i se atribuie un număr de ordine (memorat în înregistrare) care identifică poziția fizică a înregistrării în fișier.

2) *De comenzi* (cu extensia .CMD) care conțin secvențe de comenzi dBASE organizate în proceduri și corespund unor aplicații mai complexe. Un fișier de comenzi se poate crea și actualiza folosind editoare de texte sau prin comanda (MODIFY COMMAND). Sistemul de operare CP/M pune la dispoziție editoarele de texte simbolizate ED și WS, ultimul fiind mult mai performant. Printre comenzile unui fișier de comenzi se pot găsi comenzi de

*) În acest caz după descrierea fișierului la întrebarea adresată de sistem „INPUT DATA NOW"? utilizatorul răspunde acționînd ⟨N⟩ și ⟨CR⟩ sau numai ⟨CR⟩.

**) Sistemul dBASE admite maximul 16 fișiere deschise simultan.

apel la un alt fișier de comenzi (cu restricția că sistemul dBASE admite maxim 16 fișiere deschise simultan). Un fișier de comenzi se lansază în execuție cu comanda DO ⟨nume-fișier-comenzi⟩.

3) *Format de rapoarte* (cu extensia .FRM) create prin comandă REPORT și constiuite pe baza dialogului utilizator-sistem prin care utilizatorul descrie structura raportului.

4) *Index* (cu extensia .NDX) generat prin execuția comenzii INDEX, el conține valori ale cîmpului ales drept cheie de indexare, iar pentru fiecare dintre acestea numărul înregistrării din cadrul fișierului de date asociat. Prin indexare se asigură localizarea rapidă a înregistrărilor de dimensiuni mai mari folosind comandă FIND.

5) *Cu formate* (extensia .FMT), *cu variabile de memorie* (extensia .MEM) și *cu texte* (cu extensia .TXT). Fișierele cu variabile de memorie conțin valorile anumitor variabile înregistrate cu comanda SAVE și restaurate cu comanda RESTORE. Fișierele cu formate se definesc în legătură cu comenzile @ și activate prin comanda READ. Fișierele cu texte conțin comenzile și informațiile afișate pe ecran (ieșiri generate de sistemul dBASE precum și intrări tastate la terminal), identificarea și activarea fișierului de texte se realizează cu ajutorul comenzilor:

SET ALTERNATE TO ⟨nume fișier [.TXT]⟩
și SET ALTERNATE ON.

D. *Funcții* destinate realizării unor operații frecvent întîlnite în practică dintre care mai importante sînt:

1) VAL (⟨expresie-șir⟩), pentru a extrage caracterele unui șir (pînă la punctul zecimal) și a-l transforma în număr întreg corespunzător;

2) # extrage numărul înregistrării curente;

3) LEN (⟨expresie-șir⟩) pentru a calcula numărul de caractere din șirul rezultat prin evaluarea expresiei ⟨expresie-șir⟩;

4) @ (subșir⟩, ⟨șir⟩) pentru calcularea poziției unui subșir;

5) STR (⟨exp-num⟩, ⟨lungime⟩ [, ⟨zecimale⟩]), pentru evaluarea expresiei ⟨exp-num⟩ și conversia rezultatului într-un șir de caractere cu lungime ⟨lungime⟩;

6) CHR(⟨exp-int⟩), care are ca rezultat caracterul ASCII obținut prin evaluarea expresiei ⟨exp-int⟩;

7) ! (⟨exp-șir⟩), care are ca rezultat un șir echivalent cu cel inițial cu convertirea literelor mici în litere mari;

8) $ (⟨exp-șir⟩, ⟨poziție⟩, ⟨lungime⟩), pentru a extrage din ⟨exp-șir⟩ începînd din poziția ⟨poziție⟩ subșirul de lungime ⟨lungime⟩;

9) TRIM(⟨expr-șir⟩) pentru a elimina spațiile suplimentare de la sfîrșitul șirului rezultat din evaluarea expresiei ⟨exp-șir⟩;

10) *, pentru a testa starea înregistrării curente dacă înregistrarea este marcată pentru ștergere adică conține caracterul „*" la evaluarea funcției „*" se obține valoarea logică „TRUE", adică adevărat;

11) EOF, pentru a testa sfîrșitul de fișier;

12) FILE⟨nume-fișier⟩, pentru a testa existența unui fișier;

E. *Operatori*, limbajul dBASE admite folosirea a patru tipuri de operatori:

— *operatori aritmetici*: „+" pentru adunare; „—" pentru scădere; „×" pentru înmulțire; „/" pentru împărțire;

— *operatori relaționali*: >, <, =, >=, <=, # (diferit), $ operator pentru a compara subșiruri;

— *operatori logici* .OR., .AND.; .NOT.;

— *operatori pentru șiruri de caractere*: „+" concatenare de șiruri, „—" concatenare de șiruri cu eliminarea de spații dintre ele;

F. *Expresii* se obțin din constante, variabile, funcții operatori și eventual paranteze;

G. *Macro substituția* permite înlocuirea textuală a unei variabile de tip șir cu conținutul ei și are formatul & ⟨var-șir⟩. Se pot astfel defini o singură dată expresii, clauze sau comenzi (folosite frecvent) și utiliza ulterior în contexte diferite Exemple practice de utilizare a elementelor dBASE se găsesc în figurile paragrafelor ce urmează.

18.4. Codificarea structurilor standard de control în limbajul dBASE

În varianta lucrului cu fișiere de comenzi, conținutul acestora constituie proceduri elaborate după principiile programării structurate.

Structura liniară se constituie din succesiuni de operații sau grupuri de operații. Cînd mai multe operații se repetă de mai multe ori într-un program, ele se pot organiza într-o procedură și memoria într-un fișier de comenzi. Execuția grupului de comenzi se inițiază cu ajutorul comenzii DO în formatul:

DO ⟨nume-fișier-comenzi⟩

NOTA. Comanda DO admite maxim 16 niveluri de apeluri imbricate. La întîlnirea unei comenzi RETURN sau a sfîrșitului fișierului de comenzi se redă controlul nivelului imediat superior.

Pentru codificarea structurii alternative se folosește comanda IF în formatul:

```
IF ⟨exp-cond⟩
   ⟨comenzi-1⟩
   [ELSE
   ⟨comenzi-2⟩]
ENDIF
```

În grupurile de comenzi „comenzi-1" și respectiv „comenzi-2" pot exista inclusiv comenzii IF.

Structura alternativă multiplă se codifică astfel:

```
DO CASE
   CASE ⟨exp-cond-1⟩
        ⟨comenzi-1⟩
   CASE ⟨exp-cond-2⟩
        ⟨comenzi-2⟩
  [OTHERWISE
        ⟨comenzi⟩]
ENDCASE
```

Pentru codificarea structurii repetitive se folosește comanda DO WHILE cu formatul:

```
DO WHILE ⟨expr-cond⟩
   ⟨comenzi⟩
ENDDO
```

unde: ⟨comenzi⟩ se execută atîta timp cît evaluarea condiției ⟨exp-cond⟩ se obține valoarea logică „adevărat" și prin urmare condiția este îndeplinită.

Cu ajutorul *comenzii LOOP* se poate asigura din orice punct al secvenței ⟨comenzi⟩ revenirea la testarea condiției.

În figura 18.9 (*a*, *b* și *c*) sînt prezentate exemple practice de utilizare a structurilor standard de control într-o procedură simbolizată EVFACT, memorată într-un fișier de comenzi cu același nume și extensia .CMD.

```
TYPE EVFACT.CMD
* ***************     EVFACT.CMD  ****************
*                                                 *
*  SISTEMUL EVIDENTA FACTURILOR EMISE SI INCASATE*
*                                                 *
*              MODUL PRINCIPAL                    *
***************************************************
SET TALK OFF
STORE 0.00 TO  TVF
STORE 0.00 TO TVFRU
STORE 0.00 TO TVSF
STORE 0.00 TO TVAFRS
STORE 0.00 TO TOTINCAS
STORE 0.00 TO TOTSTORNO
STORE 0.00 TO TVSFI
STORE 0.00 TO TVSFP
STORE 0.00 TO TVFRI
STORE 0.00 TO TVFRP
STORE 0.00 TO TASRP
IF FILE (TEMP.DBF)
   DELETE FILE TEMP.DBF
ENDIF
IF FILE(INCTEMP.DBF)
   DELETE FILE (INCTEMP.DBF)
ENDIF
IF FILE (TEMPCF.DBF)
   DELETE FILE TEMPCF.DBF
ENDIF
IF FILE (TEMPF.DBF)
   DELETE FILE TEMPF.DBF
ENDIF
IF FILE (TEMPCOMP.DBF)
   DELETE FILE TEMPCOMP.DBF
ENDIF
IF FILE(TEMPFACT.DBF)
   DELETE FILE TEMPFACT.DBF
ENDIF
ERASE
@ 1,0 SAY ' SISTEMUL'
@ 2,1 SAY 'EVIDENTA FACTURILOR EMISE SI INCASATE'
@ 4,1 SAY 'ELABORAT DE UN COLECTIV DE CADRE DIDACTICE DE LA '
@ 5,1 SAY 'CATEDRA DE CONTABILITATE, A.S.E. BUCURESTI'
STORE ' ' TO OPT
STORE 'D' TO CONTINUATI
STORE '        ' TO DATAOP
DO WHILE !(CONTINUATI) = 'D'
 ACCEPT 'DATA : ZZ/LL/AA  ' TO DATAOP
 ACCEPT ' D-CORECTII , (CR) CONTINUARE ' TO CONTINUATI
 @ 6,0 SAY CHR(27)+CHR(89)
ENDDO
STORE 'D' TO CONTINUATI
DO WHILE !(CONTINUATI) = 'D'
 ERASE
```

Fig. 18.9. *a* — Exemplu de utilizare a structurilor standard de control.

18.5. Formatele principalelor comenzi dBASE

În cadrul acestui paragraf, pentru a veni în sprijinul utilizatorilor interesați în lucru cu limbajul dBASE (sub sistemul de operare CP/M), prezentăm comenzile mai importante și variante de utilizare a acestora.

În scrierea formatelor se stabilesc următoarele convenții:

⟨comenzi⟩, secvență de comenzi;

⟨șir caractere⟩, orice șir de caractere (delimitat prin apostrofuri, ghilimele sau paranteze drepte);

⟨exp-num⟩, expresie a cărei valoare este de tip numeric;

⟨exp-șir⟩, expresie a cărei valoare este de tip șir de caractere;

⟨listă-exp⟩ listă de expresii separate prin virgulă;

⟨cîmp⟩ orice nume de cîmp al unei înregistrări dintr-un fişier de date dBASE;

⟨fişier⟩ [...] pentru a desemna un nume de fişier (între paranteze drepte se trece extensia);

⟨cheie⟩ nume de cîmp folosit pentru indexarea sau expresia a cărei valoare constituie cheia de indexare;

⟨var⟩ numele unei variabile de tip numeric, logic sau şir de caractere;

⟨n⟩ constantă de tip numeric;

⟨domeniu⟩ o mulţime de înregistrări a unui fişier;

⟨exp-cond⟩ expresie cu valoare logică.

Şi acum scurte referiri cu privire la principalele comenzi dBASE.

```
@ 1,0 SAY ' ATENTIE'
? '          SISTEMUL ADMITE CA FIECARE PROCEDURA SA FIE ACTIVATA'
? '           O SINGURA DATA '
? '           (CU EXCEPTIA PROCEDURII INTRODUCERE FACTURI'
? '            CARE VA FI ACTIVATA DE DOUA ORI:'
? '                  - PENTRU INTRODUCERE FACTURI;'
? '                  - PENTRU REFUZ,STORNO ANULARE'
? '          GRUPATI DOCUMENTELE PE TIPURI DE OPERATII '
STORE ' ' TO OPT
@ 11,0 SAY ' OPTIUNI DE PRELUCRARE, '
@ 12,0 SAY '   1 INTRODUCERE FACTURI              '
@ 13,0 SAY '   2 INTRODUCERE INCASARI             '
@ 14,0 SAY '   3 CONSULTARE/EDITARE               '
@ 15,0 SAY '   4 EDITARE RAPOARTE                 '
@ 16,0 SAY '-  5 INCHIDERE LUNA                   '
@ 17,0 SAY '   6 EXPLICATII                       '
ACCEPT ' (CR) TERMINARE SESIUNE ' TO OPT
IF OPT = ' '
   STORE 'N' TO CONTINUATI
   LOOP
ENDIF
IF OPT = '1'
   STORE ' ' TO OPER
   STORE F TO SF
   ERASE
   DO WHILE .NOT. SF
    SET INTE OFF
    @ 1,1 SAY ' INTRODUCERE FACTURI '
    @ 2,1 SAY ' OPTIUNI'
    SET INTE ON
    @ 3,1 SAY ' F INTRODUCERE FACTURI NOI SAU FACTURI REFUZATE ULTERIOR'
    @ 4,1 SAY ' R INTRODUCERE REFUZURI, ANULARI REF. SAU STORNO FACTURI '
    ACCEPT ' TASTATI OPTIUNEA SAU (CR) PTR. TERMINARE ' TO OPER
    IF OPER = ' '
       STORE T TO SF
       LOOP
    ENDIF
    IF !(OPER) = 'F'
       DO FACTNRU
     ELSE
       IF !(OPER) = 'R'
          DO FACTRSA
        ELSE
          ? ' OPTIUNE ERONATA ! RELUATI VA ROG!'
          @ 5,0 SAY CHR(27)+CHR(87)
          LOOP
       ENDIF
    ENDIF
   ENDDO
ELSE
  IF OPT = '2'
     DO INCASARI
   ELSE
     IF OPT = '3'
        DO CONSED
      ELSE
        IF OPT = '4'
           DO REZULT
         ELSE
           IF OPT = '5'
              DO INCLUNA
            ELSE -
              IF OPT = '6'
```

Fig. 18.9. *b*.

```
                    DO EXPLIC
                  ELSE
                    @ 3,1 CHR(27)+CHR(87)
                    STORE 'D' TO CONTINUA1
                    LOOP
                  ENDIF 6
                ENDIF 5
              ENDIF 4
            ENDIF 3
          ENDIF 2
        ENDIF 1
      ENDDO
      ERASE
      @ 2,1 SAY 'EVIDENTA FACTURILOR EMISE SI INCASATE '
      STORE  ' ' TO OPER
      ? ' DORITI TOTALURILE LA IMPRIMANTA ? '
      ACCEPT ' DA/ NU DUPA CAZ ' TO OPER
      IF !(OPER) = 'D'
         ? ' ACTIVATI IMPRIMANTA ! '
         ACCEPT ' GATA (CR) ' TO VAR2
         SET PRINT ON
         EJECT
      ENDIF
      ?
      ? '          SESIUNE PRELUCRARE DATE '+'DIN ' + DATAOP
      ? '          ==========================================='
      ? ' VALOARE FACTURI NOI                          '+STR(TVF,12,2)
      ? ' VALOARE FACTURI REFUZATE ULTERIOR            '+STR(TVFRU,12,2)
      ? ' VALOARE STORNO FACTURI INTEGRAL              '+STR(TVSFI,12,2)
      ? ' VALOARE STORNO FACTURI PARTIAL               '+STR(TVSFP,12,2)
      ? ' ------------------------------'
      ? ' VALOARE STORNO FACTURI                       '+STR((TVSFI+TVSFP),12,2)
      ? ' ------------------------------'
      ? ' VALOARE FACTURI REFUZATE INTEGRAL            ' +STR(TVFRI,12,2)
      ? ' VALOARE FACTURI REFUZATE PARTIAL             ' + STR(TVFRP,12,2)
      ? ' ------------------------------'
      ? ' VALOARE FACTURI REFUZATE                     '+STR((TVFRI+TVFRP),12,2)
      ? ' ------------------------------'
      ? ' VALOARE ANULARI(STORNO SAU REFUZURI )      '+STR(TASRP,12,2)
      ? ' VALOARE FACTURI INCASATE                     ' + STR(TUTINCAS,12,2)
      ? ' VALOARE STORNO INCASARI                      ' + STR(TUTSTORNO,12,2)
      ? '==========================================================='
      ?
      SET CONSOLE OFF
      ?  ' SEMNATURA '
      SET CONSOLE ON
      ?
      ?
      ? ' LA REVEDERE '
      ?
      ?
      SET PRINT OFF
      QUIT
      RETURN


      A>
```

Fig. 18.9 *б.*

Comanda ? are formatul

? ⟨lista-exp⟩

Comanda @ are formatul:

@ ⟨coordonate⟩ [SAY ⟨exp⟩ [USING ⟨format ⟩]]
[GET var-GET [PICTURE ⟨format⟩]]

și permite afișarea de date formatate, pe ecran sau la imprimantă. Cînd afișarea se face pe ecran (opțiune implicată) coordonatele sînt de forma [X, Y] și exprimă linia și respectiv coloana (unde X poate lua valoare în domeniul {0,23} iar în Y în domeniul {0,79}.

Clauza SAY se folosește pentru afișarea unei expresii ⟨exp⟩. *Clauza USING* permite formatarea expresiei afișate. *Clauza GET* admisă numai în mod „ecran" permite afișarea valorii curente a variabilei ⟨ var-GET ⟩ și editarea ei prin execuția ulterioară a unei comenzi READ. *Clauza PICTURE* asociată clauzei GET specifică formatul de ditare (în lipsa acesteia este determinat de tipul și valoarea anterioară a variabilei). Comanda @ ... SAY ... este indicată pentru a formata rapoarte de afișat. Formatele specificate prin

clauzele USING şi PICTURE conţin caractere de control pentru afişarea pe ecran sau pe pagină. *Comanda ERASE* permite anularea ecranului şi stabileşte coordonatele „0, 0".

Comanda READ are formatul:

READ

şi declanşează lucru în mod ecran pentru editarea şi/sau introducerea de date corespunzător celor prezentate prin comanda @ ...SAY ...GET.

Comanda ACCEPT are formatul:

ACCEPT 'mesaj' TO ⟨var-şir⟩

şi se foloseşte pentru introducerea şirurilor de caractere (variabilele şir sînt create prin această comandă, lor li se atribuie caracterele introduse de utilizator). Dacă se testează ⟨CR⟩ variabilei i se atribuie valoarea desemnată prin un caracter spaţiu.

Comanda INPUT are formatul:

INPUT 'mesaj' TO ⟨var⟩

şi permite introducerea valorii unei expresii de orice tip într-o variabilă. Cu ajutorul acestei comenzi se creează o variabilă de tip numeric, logic sau şir de caractere căreia i se atribuie rezultatul evaluării expresiei date de utilizator.

Comanda STORE are formatul:

STORE ⟨expresie⟩ TO ⟨var⟩

şi atribuie variabilei ⟨var⟩ valoarea rezultată prin evaluarea expresiei ⟨expresie⟩. Variabila ⟨var⟩ nu trebuie să existe apriori (variabilă se creează cu această comandă).
Comanda operează numai asupra variabilelor.

Comanda CREATE are formatul:

CREATE ⟨fişier[.DBF]⟩

şi permite utilizatorului ca pe baza unui dialog cu sistemul să precizeze structura fişierului şi să-l creeze prin memorarea înregistrării cu structura asociată.

Comanda USE poate avea unul din formatele:

a) USE
b) USE ⟨fişier [.DBF]⟩ [INDEX ⟨listă-fişier [.NDX]⟩]

şi permite: *formatul a* dezactivarea fişierului activat prin USE; *formatul b* activarea unui fişier. Prin opţiunea INDEX se precizează numele fişierelor INDEX asociate pentru a obţine un acces rapid la date (vezi şi comanda INDEX). Unui fişier .DBF i se pot asocia maxim 7 fişiere .NDX (primele trebuie să fi fost create anterior). Orice modificare în fişierul .DBF produce actualizarea şi a fişierelor .NDX (excepţie face comanda PACK).

Comanda COPY are formatul:

COPY TO ⟨fişier-destinaţie⟩ [⟨domeniu⟩]
[FIELD ⟨lista-cîmp⟩] [FOR ⟨exp-cond⟩] [SDF]
[STRUCTURE] [DELIMITED] [WITH ⟨delimitator⟩]

şi permite copierea: numai a structurii unui fişier (în prezenţa clauzei STRUCTURE adică COPY TO ⟨fişier-destinaţie⟩ STRUCTURE); numai a datelor

făcînd abstracție de structură (în prezent opțiuni SDF) sau copierea fișierului în accepțiunea dBASE. Copierea fișierelor cu opțiunea SDF permite ca ⟨fișier-destinație⟩ să poată face obiectul prelucrării folosind un alt limbaj de programare care admite fișiere cu format standard ASCII.

Opțiunea DELIMITED permite separarea cîmpurilor prin virgulă. Prin opțiunea FOR exp-cod se poate condiționa copierea din fișierul sursă în fișierul destinație. *Opțiunea FIELD* precizează ce cîmpuri din structura fișierului definesc structura fișierului ⟨fișier-destinație⟩ sau ce date se copiază.

Comanda SKIP are formatul:

SKIP {+/−}[⟨n⟩]

și permite efectuarea unui salt înainte (+) sau înapoi (—) în raport cu înregistrarea curentă (precizată de către pointer).

Comanda INDEX are formatul:

INDEX ON ⟨cheie⟩ TO ⟨fișier [.NDX]⟩

și permite construirea unui fișier index asociat fișierului activ. Ordinea fizică a înregistrărilor din fișierul activ rămîne neschimbată. Fișierul Index (.NDX) va conține „pointeri" către înregistrările din fișierul activ (.DBF) privite, prin prisma fișierului index înregistrările sînt ordonate în funcție de valorile crescătoare ale cîmpului ⟨cheie⟩ unde ⟨cheie⟩ poate fi o expresie șir (în care pot apărea cîmpuri, variabile constante). La concatenarea mai multor cîmpuri în vederea constituirii cheii de tip șir pentru cîmpurile de tip numeric se va aplica funcția STR.

Cheia poate fi desemnată și sub forma unei expresii numerice caz în care asupra cîmpului de șir (care ar participa eventual la constituirea cheii) se aplică funcția VAL.

Prin indexare se dă posibilitatea accesului rapid la anumite înregistrări din fișierul DBF. Pentru orice modificare operată în fișierul de date sistemul asigură actualizarea fișierului index.

Comanda FIND are formatul:

FIND ⟨șir-caractere⟩

lucrează numai pe fișiere indexate și permite localizarea rapidă a înregistrărilor dintr-un fișier activat cu comanda USE ... INDEX ... (dacă fișierului activ i se asociază mai multe fișiere index localizarea se face numai în funcție de fișierul index principal). Cînd cheia este șir de caractere se poate ca prin FIND să se precizeze unul sau mai multe caractere pentru localizare.

În cazul regăsirii în funcție de conținutul unei variable aceasta se specifică în FIND prin procedeul macrosubstituției (&⟨var⟩). Asupra înregistrării astfel localizate se poate acționa cu orice operație permisă de sistem.

Comanda LOCATE are formatul:

LOCATE ⟨domeniu⟩ FOR ⟨exp-cond⟩

și permite localizarea primei înregistrări din ⟨domeniu⟩ care satisface condiția ⟨exp-cond⟩.

Comanda CONTINUE are formatul:

CONTINUE

și permite continuarea căutării inițiate prin LOCALE în fișierul activ folosind criteriile de căutare precizate prin LOCATE.

Comanda APPEND poate avea unul din formatele:

a) APPEND
b) APPEND BLANK
c) APPEND FROM ⟨ fișier ⟩ FOR ⟨ exp-cond ⟩
[SDF][DELIMITED]

și permite: (a) adăugarea de înregistrări în mod interactiv: sistemul afișează structura (numele cîmpurilor) înregistrării fișierului activ iar utilizatorul înregistrează datele. Procesul se încheie cînd se tastează ⟨ CR ⟩ ca prim caracter al primului cîmp dintr-o nouă înregistrare; (b) permite adăugarea unei singure înregistrări care conține spații; (c) permite adăugarea tuturor înregistrărilor preluate din fișier sau numai a celor care îndeplinesc condiția precizată prin ⟨ exp-cond ⟩.

Comanda REPLACE are formatul:

REPLACE ⟨ domeniu ⟩ [FOR ⟨ exp-cond ⟩] ⟨ cîmp-1 ⟩
WHITH ⟨ exp-1⟩, [⟨ cîmp-2 ⟩ WITH ⟨ exp-2 ⟩] ...
[⟨ cîmp-n ⟩ WITH ⟨ exp-n ⟩]

și permite modificarea conținutului cîmpurilor fișierului activ. Valorile cîmpurilor ⟨ cîmp-1 ⟩, ..., ⟨ cîmp-n ⟩ sînt înlocuite cu cele rezultate din evaluarea expresiilor ⟨ exp-1 ⟩ ... ⟨ exp-n ⟩. Dacă se operează modificări asupra unui cîmp de indexare se face actualizarea și a fișierului index (dacă activarea fișierului s-a făcut cu formatul USE ... INDEX ...)

Comanda SORT are formatul:

SORT ON ⟨ cheie ⟩ TO ⟨ destinație ⟩ [ASCENDING][DESCENDING] și creează un fișier destinație cu aceleași înregistrări ale fișierului activ cu precizarea că înregistrările sînt sortate în ordine crescătoare (ASCENDING) sau descrescătoare (DESCENDING). Clauza ASCENDING este implicită. Fișierul sursă rămîne activ în continuare cheile de sortare pot fi și de tip ⟨ șir-de-caractere ⟩. Un fișier poate fi sortat după mai multe chei prin sortări succesive.

Comanda DISPLAY poate avea unul din formatele:

a) DISPLAY STRUCTURE pentru a afișa structura unui fișier DBF;

b) DISPLAY MEMORY, pentru a afișa valorile curente ale variabilelor de memorie;

c) DISPLAY FILES ON ⟨ unitate-disc ⟩ [LIKE ⟨ specificație-fișier ⟩], pentru a afișa numele fișierelor de pe discul implicit sau specificat prin clauza ON sau numai ale celor specificate prin clauza LIKE.

d) DISPLAY ⟨ domeniu ⟩ [FOR ⟨ exp-cond ⟩ [⟨ listă-exp ⟩] [OFF] pentru obținerea de rapoarte asupra datelor din fișierul activ (fără definirea antetului și nici efectuarea de prelucrări). Clauza OFF inhibă scrierea numărului de înregistrare. Listarea marchează pauze după grupuri de 15 înregistrări.

Comanda LIST este asemănătoare comenzii DISPLAY cu precizarea că valoarea implicită pentru (domeniu) este ALL iar listarea este continuă.

Comanda DELETE poate avea unul din formatele:

a) DELETE ⟨ domeniu ⟩ [FOR ⟨ exp-cond ⟩]
b) DELETE FILE ⟨ fișier ⟩

și asigură: (a) invalidarea (marcarea pentru ștergere) a înregistrărilor care îndeplinesc condiția ⟨ exp-cond ⟩ extrase din ⟨ domeniu ⟩. În lipsa domeniului efectul se rezumă la înregistrarea curentă (în starea invalidat o înregistrare

nu poate fi decît afișată); (b) ștergerea unui fișier și eliberarea spațiului ocupat (sistemul nu permite ștergerea unui fișier activ). Dezactivarea unui fișier se realizează prin comanda USE.

Notă. Marcajul de invalidare este afișat prin caracterul '*' și apare după numărul înregistrări.

Comanda PACK permite eliminarea fizic a înregistrărilor marcate prin DELETE.

Comanda UPDATE are formatul:

UPDATE FROM ⟨ sursă ⟩ ON ⟨ cheie ⟩
[ADD ⟨ listă-cip-num ⟩]
[REPLACE ⟨ listă cîmp ⟩]

și permite actualizarea fișierului de date activ (care trebuie să fie sortat sau indexat după valorile cîmpului cheie cu valori extrase din fișierul sursă în baza valorilor cîmpului cheie care le este comun și are același nume, lucru valabil și pentru cîmpurile precizate prin ADD).

Comanda GOTO poate avea unul din formatele:

a) GOTO RECORD ⟨n⟩
b) GOTO TOP
c) GOTO BOTTOM
d) GOTO exp-num-int

și permite poziționarea pointerului: (a) pe înregistrarea cu numărul ⟨n⟩; (b) pe prima înregistrare a fișierului; (c) pe ultima înregistrare a fișierului; (d) pe o înregistrare al cărui număr (întreg) rezultă din evaluarea expresiei ⟨ exp-num-int ⟩.

Comanda EDIT are formatul:

EDIT ⟨n⟩

și permite modificarea conținutului cîmpurilor din înregistrările fișierului activ în mod selectiv.

Comanda SELECT are formatul:

SELECT [PRIMARY][SECONDARY]

și permite precizarea zonei în care se activează fișierul (PRIMARY — primară sau SECONDARY — secundară). Zona primară este selectată implicit. SELECT constituie deci un comutator între cele două zone. Comenzile GOTO; SKIP, COPY, DISPLAY, REPLACE, REPORT, SORT, INDEX, APPEND etc. afectează numai fișierul selectat. Expresiile cu părți componente ale unei comenzi se pot constitui folosind date din ambele zone, precizate dacă sînt identificate prin P. respectiv S. Același fișier poate fi acționat în ambele zone.

Comanda JOIN are formatul*):

JOIN TO ⟨ destinație ⟩ FOR ⟨ ex-pcond ⟩
[FIELDS ⟨ listă-cîmp ⟩]

și permite reuniunea condiționată (unu la unu) a înregistrărilor din două fișiere, unul în primar iar altul în secundar, într-un fișier destinație cu struc-

*) Secvența de comenzi care precede comanda JOIN este:
SELECT PRIMARY
USE fișier-primar
SELECT SECONDARY
USE fișier-secundar
SELECT PRIMARY

tura obținută prin concatenarea celor două fișiere (secundar și primar) sau conform celor specificate prin FIELDS.

Comanda SUM are formatul:

SUM ⟨ listă-exp-num ⟩ [⟨ domeniu ⟩]
FOR ⟨ exp-cond ⟩ [TO ⟨ listă-var ⟩

și permite însumarea expresiilor numerice formate din maxim 5 elemente) folosind cîmpuri din înregistrări și depune opțional rezultatul în variabile. Variabilele din listă-var nu trebuie să existe apriori. Rezultatele din evaluare list-exp-num sînt afișate pe aceeași linie.

Comanda TOTAL are formatul:

TOTAL ON ⟨ cheie ⟩ TO ⟨ destinație ⟩
[FIELDS ⟨ listă-cîmp ⟩] [FOR ⟨ exp-cond ⟩]

și permite crearea unui fișier prin însumarea datelor din fișierul activ (care trebuie indexat sau sortat după cîmpul ⟨ cheie ⟩) pentru cîmpurile precizate prin FIELDS ale înregistrărilor care au aceeași valoare pentru cîmpul ⟨ cheie ⟩. Pentru fiecare valoare a cîmpului ⟨ cheie ⟩ se constituie cîte o înregistrare în fișierul destinație care conține cheia plus valorile obținute prin însumare pentru cîmpurile precizate prin FIELDS (dacă aceasta este prezentă).

În lipsa opțiunii FIELDS structura fișierului destinație este identică cu cea a fișierului activ iar valorile cîmpurilor neînsemnate sînt cele corespunzătoare primei înregistrări din grupă.

Comanda REPORT are formatul:

REPORT [FORM ⟨ fișier [.FRM] ⟩] [FOR ⟨ exp-cond ⟩]
[⟨ domeniu ⟩] [TO PRINT]

și permite construirea de rapoarte pe baza datelor conținute în fișierul activ. Prin clauza TO PRINT se comandă tipărirea raportului și la imprimantă. Valoarea implicită pentru domeniu este ALL. Clauza FORM precizează crearea fișierului în care se memorează formatul raportului sau utilizarea unui astfel de fișier creat anterior. Raportarea se poate face „în detaliu" sau „în rezumat" (adică numai cu totalurile și subtotalurile obținute prin însumare.

Comanda RETURN are formatul:

RETURN

și încheie execuția unui fișier de comenzi (controlul este redat fișierului de comenzi la nivel imediat superior sau terminalului după caz).

Comanda QUIT are formatul:

QUIT [TO '⟨ lista-com ⟩']

și încheie o sesiune de lucru în dBASE II redînd controlul sistemului de operare (totodată se închid toate fișierele deschise). ⟨ Listă-cond ⟩ poate avea maxim 254 de caractere (prin care se lansează comenzi CP/M sau se relansează dBASE).

Comanda SET poate avea unul din formatele:

a) SET ⟨ par ⟩ ON/OFF
b) SET ⟨par⟩ TO ⟨opțiune⟩

unde *Formatul a* permite comutarea valorilor ON/OFF parametrilor ⟨ par ⟩ din care prezentăm:

TALK ON rezultatele executării comenzilor sînt afișate pe ecran;
OFF nu afișează rezultatele executării comenzilor la terminal;

PRINT ON informațiile de ieșire pe ecran sînt imprimate și la imprimantă
OFF tipărirea la imprimantă este inhibată.

— *Formatul b* permite atribuirea de valori diferite de cele implicate. Spre exemplu: SET HEADING TO ⟨ şir ⟩ specifică faptul că antetul de pagină specificat prin ⟨ şir ⟩ este tipărit prin comanda REPORT la începutul fiecărei pagini a raportului: SET FORMAT TO PRINT precizează că rezultatul comenzilor este destinat imprimantei. Unele exemple de utilizare a comenzilor în regim interactiv sînt prezentate în figurile 18.10, 18.11 şi 18.12.

```
. USE PERSONAL
.
.
. COPY STRUCTURE TO PERS1
. USE PERS1
. DISPLAY STRU
STRUCTURE FOR FILE:  PERS1.DBF
NUMBER OF RECORDS:   00000
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD      NAME        TYPE WIDTH   DEC
001      LOC:MCA      N    004
002      MARCA        N    005
003      NUME         C    015
004      PRENUME      C    015
005      FUNCTIA      C    002
006      RETRIB       N    004
007      AN:NAST      N    006
008      AN:ANGAJ     N    006
009      STARE:CIV    C    001
010      NR:COPII     N    001
011      SEX          C    001
** TOTAL **                 00061
```

Fig. 18.10. — Copierea structurii unui fişier.

```
. USE PERSONAL
. REPLACE ALL RETRIB WITH RETRIB * 0.05
00010 REPLACEMENT(S)
. LIST
00001   1011   4131 POPESCU        NICOLAE      EC   200  130347  150270 C  2 M
00002   1011   4151 IONESCU        ION          EC   190  120249  110372 C  1 M
00003   1011   4171 SLAVESCU       MARIA        IN   165  110453  130178 C  1 F
00004   1011   4191 GEORGESCU      ION          FU   125  130749  120772 C  1 M
00005   1022   4301 ALBESCU        VASILE       EC   142  120558  110880 C  0 M
00006   1022   4501 DUMITRU        ION          IN   222   10147   20270 N  0 M
00007   1033   4541 RADU           MIRCEA       EC   175  121158  110280 C  0 M
00008   1033   4571 BIDASCU        CARMEN       EC   185  120157  130480 C  1 F
00009   1011   1234 IONESCU        VASILE       EC   228  120349  120569 C  1 M
00010   1234   1234 VASILESCU      EUGENIU      EC   260  120338  120459 C  2
```

Fig. 18.11. — Modificarea conţinutului înregistrărilor unui fişier (exemplul 1).

```
. REPL ALL RETRIB WITH RETRIB + (RETRIB * 0.05)
00010 REPLACEMENT(S)
. LIST
00001   1011   4131 POPESCU        NICOLAE      EC   210  130347  150270 C  2 M
00002   1011   4151 IONESCU        ION          EC   199  120249  110372 C  1 M
00003   1011   4171 SLAVESCU       MARIA        IN   173  110453  130178 C  1 F
00004   1011   4191 GEORGESCU      ION          FU   131  130749  120772 C  1 M
00005   1022   4301 ALBESCU        VASILE       EC   149  120558  110880 C  0 M
00006   1022   4501 DUMITRU        ION          IN   233   10147   20270 N  0 M
00007   1033   4541 RADU           MIRCEA       EC   183  121158  110280 C  0 M
00008   1033   4571 BIDASCU        CARMEN       EC   194  120157  130480 C  1 F
00009   1011   1234 IONESCU        VASILE       EC   239  120349  120569 C  1 M
                          SCU      EUGENIU      EC   273  120338  120459 C  2 M
```

Fig. 18.12. — Modificarea conţinutului înregistrărilor unui fişier (exemplul 2).

18.6. Exemple practice dezvoltate de proceduri dBASE

Aşa cum am văzut comenzile dBASE se pot utiliza în regim interactiv sau prin memorare în fişiere de comenzi.

Dacă prima variantă este relativ simplă şi accesibilă la nivel utilizator (ceea ce a făcut să se avanseze conceptul de programare la nivel utilizator), cea de-a doua respectă conceptele programării convenţionale cu însemnatele facilităţi oferite de limbajul dBASE pe linia descrierii rapoartelor, regăsirii datelor şi codificării structurilor standard de control.

Procedura din figura 18.14 (I II, III IV, V, VI) conține comenzile dBASE pentru verificarea corectitudinii și înregistrarea datelor în fișierele B:FACTURI (unde B: precizează că fișierul este memorat pe o unitate de volum cu folosirea unității B: de discuri) și B:COMPFACT. Structura înregistrărilor corespunzătoare celor două fișiere este prezentată în figurile 18.13 *a* și 18.13 *b*.

```
    DBASE
USE B:FACTURI
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:FACTURI.DBF
NUMBER OF RECORDS:   00005
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD      NAME       TYPE WIDTH   DEC
001      NR:FACT     N    005
002      DATA:FACT   N    004
003      AVIZ:EXP    N    004
004      CLIENT      C    015
005      BANCA       C    015
006      JUDET       C    002
007      STARE       C    001
** TOTAL **          00049
.
```

Fig. 18.13 *a*. — Structura fișierului B:FACTURI.

```
USE B:COMPFACT
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:COMPFACT.DBF
NUMBER OF RECORDS:   00010
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD      NAME       TYPE WIDTH   DEC
001      NR:FACT     N    005
002      DEN:ELEM    C    004
003      CANT:EI     N    005
004      VAL:EI      N    012    002
005      CANT:ER     N    005
006      VAL:ER      N    012    002
007      CANT:I      N    005
008      VAL:I       N    012    002
009      MOTIV:R     C    001
** TOTAL **          00062
```

Fig. 18.13 *b*. — Structura fișierului B:COMPFACT.

```
A>TYPE FACTNRU.CMD
************************  FACTNRU.CMD  ********************
*
*       MODULUL FACTNRU.CMD PERMITE UTILIZATORULUI
*       CA PE BAZA UNUI DIALOG CU CALCULATORUL SA
*       INCARCE BAZA DE DATE PRIVIND FACTURILE EMISE
*************************************************************
SET TALK OFF
USE B:FACTURI
COPY TO TEMPFACT STRUCTURE
USE B:COMPFACT
COPY TO TEMPCOMP STRUCTURE
USE TEMPCOMP
SELECT SECO
USE TEMPFACT
SELECT PRIM
STORE ' ' TO OPER
STORE F TO SF
DO WHILE .NOT. SF
 ERASE
 @ 1,1 SAY 'OPTIUNI DE PRELUCRARE '
 @ 2 ,1 SAY '  F  FACTURI NOI
 @ 3,1 SAY  '  U  FACTURI REFUZATE ULTERIOR '
 @ 4,1 SAY 'ATENTIE ! MARCATI (CR) DUPA INTRODUCEREA ';
           +' DATELOR DIN TOATE FACTURILE'
 ACCEPT ' TASTATI U,F SAU (CR) PENTRU SFIRSIT ' TO OPER
 IF OPER = ' '
   STORE T TO SF
   LOOP
 ENDIF
 IF !(OPER) = 'F' .OR. !(OPER) = 'U'
    ? '  PREGATITI INTRODUCEREA DATELOR '
    STORE 0 TO CONTOR
    DO WHILE CONTOR < 80
     STORE CONTOR + 1 TO CONTOR
    ENDDO
  ELSE
    ? ' OPTIUNE ERONATA ! RELUATI VA ROG ! '
    LOOP
 ENDIF
 SET INTE OFF
 IF !(OPER) = 'F'
    @ 2,6 SAY 'FACTURI NOI'
  ELSE
    IF !(OPER) = 'U'
      @ 3,6 SAY 'FACTURI REFUZATE ULTERIOR'
    ENDIF
 ENDIF
 SET INTE ON
```

Fig. 18.14, I. — Organizarea procedurii FACTNRU.CMD.

```
STORE 'D' TO CORECT
DO WHILE !(CORECT) = 'D'
 STORE 0.00 TO VALFACT
 STORE '      ' TO MNR:FACT
 STORE '     ' TO  MAVIZ:EXP
 STORE '       ' TO MDATA:FACT
 STORE '                    ' TO MBANCA
 STORE '  ' TO MJUDET
 STORE '                    ' TO MCLIENT
 STORE 'D' TO CORECT
 @ 8,5 SAY 'NUMAR FACTURA ? ' GET MNR:FACT PICTURE '99999'
 @ 8,30 SAY 'DATA EMITERII FACTURII :ZZLLAA ';
                GET MDATA:FACT PICTURE '999999'
 @ 9,5 SAY 'AVIZ EXPEDITIE ? ' GET MAVIZ:EXP
 @ 9,40 SAY 'CLIENT ? ' GET MCLIENT
 @ 10,5 SAY 'BANCA ? ' GET MBANCA
 @ 10,40 SAY 'JUDETUL ? ' GET MJUDET
 READ
 STORE VAL(MNR:FACT) TO MNR:FACT
 @ 8,20 SAY MNR:FACT
 @ 8,62 SAY MDATA:FACT
 @ 9,23 SAY MAVIZ:EXP
 @ 9,50 SAY MCLIENT
 @ 10 ,12 SAY MBANCA
 @ 10,51 SAY MJUDET
 STORE 0 TO FLAG
 IF MNR:FACT = 0
    ? ' NR. FACTURA  = 0 '
    STORE 1 TO FLAG
 ENDIF
 IF VAL($(MDATA:FACT,1,2)) < 1 .OR. VAL($(MDATA:FACT,1,2)) > 31
    .OR. VAL($(MDATA:FACT,3,2)) < 1 .OR. VAL($(MDATA:FACT,3,2)) > 12
    .OR. VAL($(MDATA:FACT,5,2)) < 83
    STORE 1 TO FLAG
    ? ' DATA ERONATA ? '
 ENDIF
 IF !(MCLIENT)  = '                    '
    ? ' DENUMIRE CLIENT = SPATII ? '
    STORE 1 TO FLAG
 ENDIF
 IF !(MBANCA) = '                    '
    STORE 1 TO FLAG
    ? ' BANCA = SPATII ? '
 ENDIF
 IF !(MJUDET) = '  '
    STORE 1 TO FLAG
    ? ' JUDET = SPATII ? '
 ENDIF
 IF FLAG = 1
    ? CHR(20)+CHR(66)+ 'VALORILE DATELOR INTRODUSE SINT ERONATE';
    + CHR(20)+CHR(68)+CHR(97)+CHR(07)
 ENDIF
 ACCEPT ' D PTR. CORECTARE SAU (CR) CONTINUARE' TO CORECT
 @ 11,0 SAY CHR(27)+CHR(87)
ENDDO
IF FLAG = 1
   LOOP
ENDIF
STORE $(MDATA:FACT,5,2)+$(MDATA:FACT,3,2)+$(MDATA:FACT,1,2) TO MDATAIFACT
STORE 11 TO LINIE
STORE 'D' TO VGRUPEL
DO WHILE VGRUPEL = 'D'
 STORE 0.00 TO VALGRUP
 STORE 0 TO CANTGRUP
 STORE LINIE + 1 TO LINIE
 IF LINIE > 23
    STORE 11 TO LINIE
    @ LINIE,0 SAY CHR(27)+CHR(87)
```

Fig. 18,14, II.

```
ENDIF
STORE '      ' TO MDENELEM
@ LINIE,1 SAY ' ELEMENT ? ' GET MDENELEM
READ
@ LINIE,13 SAY MDENELEM
STORE ' ' TO CORECTIEL
ACCEPT ' D-CORECTII (CR) CONTINUARE ' TO CORECTIEL
IF !(CORECTIEL) = 'D'
   @ LINIE,0 SAY CHR(27)+CHR(87)
   LOOP
ENDIF
@ (LINIE + 1),0 CHR(27)+CHR(87)
IF !(MDENELEM) = '
   STORE 0 TO CONTOR
   @ LINIE,40 SAY ' CORECTII ? '
   DO WHILE CONTOR < 80
    STORE CONTOR + 1 TO CONTOR
   ENDDO
   @ LINIE,0 SAY CHR(27)+CHR(87)
   LOOP
ENDIF
STORE 'D' TO VCOR4
DO WHILE !(VCOR4) = 'D'
 STORE 'D' TO VCOR3
 DO WHILE !(VCOR3) = 'D'
  STORE '      ' TO MCANT
  STORE 0.00 TO MVAL
  @ LINIE,0 SAY CHR(27)+CHR(87)
  @ LINIE,0 SAY 'ELEMENT '+MDENELEM
  @ LINIE,30 SAY 'CANTITATE ? ' GET MCANT PICTURE '999999'
  @ LINIE,50 SAY 'VALOARE ? ' GET MVAL PICTURE '99999999999'
  READ
  STORE VAL(MCANT) TO MCANT
  STORE 0 TO FLAG
  @ LINIE,40 SAY MCANT
  @ LINIE,63 SAY MVAL
  IF MVAL = 0 .AND. MCANT # 0
     ? ' DATE PENTRU VALOARE ? '
     STORE 1 TO FLAG
  ENDIF
  STORE LINIE + 1 TO LINIE
  IF LINIE > 23
     STORE 11 TO LINIE
     @ LINIE,0 SAY CHR(27)+CHR(87)
  ENDIF
  ACCEPT ' D-CORECTII (CR) CONTINUARE ' TO VCOR3
  @ LINIE,0 SAY CHR(27)+CHR(87)
  STORE LINIE - 1 TO LINIE
  IF FLAG = 1
     STORE 'D' TO VCOR3
  ENDIF
 ENDDO
 IF MCANT = 0 .AND. MVAL = 0
    STORE 'N' TO VCOR4
    IF VALGRUP = 0
       STORE LINIE + 1 TO LINIE
       IF LINIE > 23
          STORE 11 TO LINIE
          @ LINIE,0 SAY CHR(27)+CHR(87)
```

Fig. 18,14, III.

```
      ENDIF
      ? ' VALOARE = 0 ? NU SE PRELUCREAZA'
      STORE 0 TO CONTOR
      DO WHILE CONTOR < 70
       STORE CONTOR + 1 TO CONTOR
      ENDDO
      @ LINIE,0 CHR(27)+CHR(89)
      STORE LINIE - 1 TO LINIE
    ELSE
      STORE VALGRUP + VALFACT TO VALFACT
      SELE PRIM
      APPEND BLANK
      REPL NR:FACT WITH NNR:FACT, DEN:ELEM WITH MDENELEM
      REPL VAL:EI WITH VALGRUP,CANT:EI WITH CANTGRUP,;
                CANT:ER WITH CANTGRUP,VAL:ER WITH VALGRUP
      IF !(OPER) = 'F'
         REPL MOTIV:R WITH 'F'
       ELSE
         REPL MOTIV:R WITH 'U'
      ENDIF
      STORE LINIE + 1 TO LINIE
      IF LINIE > 22
         STORE 11 TO LINIE
         @ LINIE,0 CHR(27)+CHR(87)
      ENDIF
      @ LINIE,0 SAY ' TOTAL : CANTITATE '+STR(CANTGRUP,5);
        + ' VALOARE' +STR(VALGRUP,12,2)
    ENDIF
    STORE LINIE + 1 TO LINIE
    IF LINIE > 23
       STORE 11 TO LINIE
       @ LINIE,0 CHR(27)+CHR(87)
    ENDIF
    ACCEPT ' MAI AVETI ELEMENTE ? D/N ? ' TO VGRUPEL
    @ LINIE,0 SAY CHR(27) + CHR(89)
    STORE LINIE - 1 TO LINIE
  ELSE
    STORE MVAL + VALGRUP TO VALGRUP
    STORE MCANT + CANTGRUP TO CANTGRUP
    STORE LINIE + 1 TO LINIE
    IF LINIE > 23
       STORE  11 TO LINIE
       @ LINIE,0 SAY CHR(27)+CHR(87)
    ENDIF
  ENDIF
ENDDO VGRUPEL
@ 11,0 SAY CHR(27)+CHR(87)
@ 11,0 SAY ' TOTAL VALOARE FACTURA '+STR(VALFACT,12,2)
STORE 0 TO CONTOR
DO WHILE CONTOR < 80
 STORE CONTOR + 1 TO CONTOR
ENDDO
```

Fig. 18.14. IV.

```
SELECT SECO
APPEND BLANK
REPL NR:FACT WITH NNR:FACT, DATA:FACT WITH VAL(NDATA:FACT),;
      AVIZ:EXP WITH VAL(NAVIZ:EXP), CLIENT WITH NCLIENT
REPL BANCA WITH NBANCA, JUDET WITH NJUDET
IF !(OPER) = 'F'
   STORE TVF + VALFACT TO TVF
 ELSE
   STORE TVFRU + VALFACT TO TVFRU
ENDIF
ENDDO
USE B:COMPFACT INDEX B:INDCOMP
APPEND FROM TEMPCOMP
USE B:FACTURI INDEX B:INDFACT
APPEND FROM TEMPFACT
SELE PRIM
USE
SELE SECO
USE
USE B:OPFACT
APPEND FROM TEMPCOMP
USE B:OPFACT
REPL TIP:OP WITH 1 FOR TIP:OP = 0
USE
ERASE
@ 1,0 SAY ' DORITI O LISTA A OPERATIILOR EFECTUATE ASTAZI ? '
STORE ' ' TO RASPUNS
ACCEPT 'D-DACA DA ,(CR) DACA NU ' TO RASPUNS
IF !(RASPUNS) = 'D'
   IF FILE(INDCOMP.NDX)
      DELETE FILE INDCOMP.NDX
   ENDIF
   USE TEMPCOMP
   INDEX ON STR(NR:FACT,5) TO INDCOMP
   SELE PRIM
   USE TEMPFACT
   SELE SECO
   USE TEMPCOMP INDEX INDCOMP
   SELE PRIM
   ? ' ATENTIE ! ACTIVATI IMPRIMANTA ! '
   ACCEPT ' DATA ? DACA DA (CR) ' TO VVCOR1
   SET PRINT ON
   EJECT
   ? '   CENTRALIZATORUL FACTURILOR EMISE SAU REFUZATE ULTERIOR'
   ?
   ? '   SESIUNE DE LUCRU DIN DATA '+DATAOP
   ?
   ? '=================================================================='
   ? ' NR.FACT  DATA   AVIZ  CLIENT              BANCA            JUDET'
   ? '------------------------------------------------------------------'
   ? ' DEN. ELEM   CANTITATE  VALOARE  '
   ? '==================================================================='
   DO WHILE .NOT. EOF
    ? '  ' + STR(NR:FACT,5) + '    ' + STR(DATA:FACT,6) + '  ';
        +STR(AVIZ:EXP,4) + '  '+ CLIENT +'  '+BANCA +'  '+JUDET
```

Fig. 18.14, V.

```
?
STORE STR(NR:FACT,5) TO KEY
SELE SECO
FIND '&KEY'
IF # = 0
   ? ' COMPONENTE FACTURA LIPSA  ????? '
   STORE 0 TO CONTOR
   DO WHILE CONTOR < 70
    STORE CONTOR + 1 TO CONTOR
   ENDDO
 ELSE
   LOCATE FOR NR:FACT = P.NR:FACT
   STORE 0.00 TO VALFACT
   DO WHILE .NOT. EOF
    ? '  ' +DEN:ELEM + '        '+STR(CANT:EI,5) + '        ';'
          +STR(VAL:EI,12,2)
    STORE VALFACT + VAL:EI TO VALFACT
    ? '-----------------------------------------------------'
    CONTINUE
   ENDDO
   ? ' TOTAL FACTURA = ' + STR(VALFACT,12,2)
   ? '====================================================='
 ENDIF
 SELE PRIM
 SKIP
ENDDO
?
? '=========================================================='
? ' VALOARE FACTURI EMISE = ' +STR(TVF,12,2)
? '----------------------------------------------------'
? ' VALOARE FACTURI REFUZATE ULTERIOR '  +STR(TVFRU,12,2)
? '=========================================================='
? ' TOTAL VALOARE FACTURI ADAUGATE  '+ STR((TVF+ TVFRU),12,2)
? ' ========================================================'
SET CONS OFF
? ' SEMNATURA '
?
SET CONS ON
ENDIF
SET PRINT OFF
RETURN
```

Fig. 18.14, VI.

La execuția procedurii efectul ultimelor instrucțiuni, în varianta listării operațiilor și la imprimantă, este prezentat în figura 18.15. Acest exemplu constituie și o variantă de editare rapoarte fără folosirea comenzii REPORT și-n consecință fără descrierea prealabilă a structurii raportului.

În figura 18.16 se prezintă structura fișierului simbolizat B:OPFACT care conține istoricul facturilor.

```
CENTRALIZATORUL FACTURILOR EMISE SAU REFUZATE ULTERIOR

SESIUNE DE LUCRU DIN DATA 12/04/85

==========================================================================
NR.FACT  DATA   AVIZ  CLIENT              BANCA              JUDET
--------------------------------------------------------------------------
DEN. ELEM   CANTITATE  VALOARE
==========================================================================
 18900    850412  1234  I.A.S TULCEA      BAIA TULCEA        TL

 2345        45          89000.00
--------------------------------------------------------------------------
 3456        89          89000.00
--------------------------------------------------------------------------
TOTAL FACTURA =     178000.00
==========================================================================

==========================================================================
VALOARE FACTURI EMISE =     178000.00
--------------------------------------------------------------------------
VALOARE FACTURI REFUZATE ULTERIOR          0.00
==========================================================================
TOTAL VALOARE FACTURI ADAUGATE      178000.00
==========================================================================
SEMNATURA
```

Fig. 18.15. — Centralizator factori emise sau refuzate ulterior obținut prin execuția procedurii din figura 18.14.

Procedura din figura 18.17 are ca obiectiv înregistrarea în baze de date privind facturile operațiilor referitoare la refuzuri, stornări sau anulări (refuzuri/stornări). Refuzurile și stornările pot fi parțiale sau integrale. Pentru

refuzuri se precizează și motivul refuzului. Procedura realizează și verificarea corectitudinii datelor dînd utilizatorului posibilitatea de a face corecții (la cerere).

Editarea la imprimantă a jurnalului operațiilor efectuate s-a prevăzut să se realizeze la cererea utilizatorului, odată cu introducerea datelor în sistem.

```
USE B:OPFACT
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:OPFACT.DBF
NUMBER OF RECORDS:   00084
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD      NAME       TYPE WIDTH   DEC
001      TIP:OP      N    001
002      NR:FACT     N    005
003      DATA:FACT   N    006
004      DEN:ELEM    C    004
005      CANT:ER     N    005
006      VAL:ER      N    012    002
007      MOTIV:R     C    001
** TOTAL **                00035
```

Fig. 18.16. — Structura fișierului B:OPFACT.

```
TYPE FACTRSA.CMD
********************* FACTRSA.CMD ***********************
*                                                       *
*    MODULUL FACTRSA.CMD PERMITE OPERAREA IN BAZA DE    *
*    DATE A OPERATIILOR PRIVIND : REFUZURILE, STORNARILE*
*    SI ANULARILE DE REFUZURI/STORNARI.                 *
*******************************************************
SET TALK OFF
USE B:OPFACT
COPY TO TEMP STRU
 SELE PRIM
USE TEMP
SELE SECO
USE B:COMPFACT INDEX B:INDCOMP
SELE PRIM
ERASE
STORE ' ' TO IMPRIM
? ' DORITI IMPRIMAREA DATELOR PRIVIND OPERATIILE'
? ' SESIUNII CURENTE ? '
ACCEPT ' D-DACA DA , (CR) IN CAZ CONTRAR ' TO IMPRIM
IF !(IMPRIM) = 'D'
   ? ' ACTIVATI IMPRIMANTA ! '.
   ACCEPT ' GATA  ? DACA DA (CR) ' TO RASP
   SET PRINT ON
   EJECT
   ? ' CENTRALIZATOR OPERATII : STORNO,REFUZURI,ANULARI S/R '
   ?
   ? ' SESIUNEA DE LUCRU DIN :'+DATAUP
   SET PRINT OFF
ENDIF
STORE ' ' TO OPER
STORE F TO SF
DO WHILE .NOT. SF
 STORE 0.00 TO VALFACT
 ERASE
 SET INTE OFF
 @ 1,1 SAY 'OPTIUNI DE PRELUCRARE'
 SET INTE ON
 @ 2,1 SAY '  S STORNO              '
 @ 3,1 SAY '  R REFUZ               '
 @ 4,1 SAY '  A ANULARE REFUZ/STORNO '
 ACCEPT ' TASTATI S,R,A SAU (CR) PTR. TERMINARE ' TO OPER
 IF OPER = ' '
   STORE T TO SF
   LOOP
 ENDIF
 SET INTE OFF
 IF !(OPER) = 'S'
   @ 2,5 SAY 'STORNO'
 ELSE
   IF !(OPER) =  'R'
     @ 3,5 SAY  'REFUZ'
   ELSE
     IF !(OPER) = 'A'
       @ 4,5 SAY 'ANULARE REFUZ/STORNA '
     ENDIF
   ENDIF
```

Fig. 18.17.I. — Organizarea procedurii FACTRSA.CMD.

```
ENDIF
SET INTE ON
IF !(OPER) = 'S' .OR. !(OPER) = 'R' .OR. !(OPER) = '▲'
   @ 5,0 SAY CHR(27)+CHR(87)
 ELSE
   ? ' OPTIUNE ERONATA ! RELUATI VA ROG ! '
   @ 5,0 SAY CHR(27)+CHR(87)
   LOOP
ENDIF
STORE 'D' TO VCOR1
DO WHILE !(VCOR1) = 'D'
 @ 9,0 SAY CHR(27)+CHR(87)
 STORE ' ' TO MREFSTOR
 STORE '      ' TO MNR:FACT
 STORE '        ' TO MDATA:FACT
 STORE ' ' TO MMOTIVREF
 @ 9,0 SAY 'NUMAR FACTURA ' GET MNR:FACT PICTURE '99999'
 @ 9,40 SAY 'DATA OPERATIE: ZZLLAA ';
                          GET MDATA:FACT PICTURE '999999'
 IF !(OPER) = 'R' .OR. !(OPER) = 'S'
    @ 10,0 SAY 'REFUZ/STORNO P-PARTIAL 1-INTEGRAL ';
         GET MREFSTOR
 ENDIF
 IF !(OPER) = 'R'
    @ 11,0 SAY 'MOTIV REFUZ ' GET MMOTIVREF
    @ 5,1 SAY ' COD  MOTIV REFUZ          COD  MOTIV REFUZ
    @ 6,1 SAY ' 1 FINANTARE NEDESCHISA    4 MARFA PRIMITA IN AVANS'
    @ 7,1 SAY ' 2 LIPSA MIJLOACE BANESTI 5 MARFA EXPEDIATA INCOMPLETA'
    @ 8,1 SAY ' 3 MARFA NEPRIMITA         6 ALTE CAUZE
 ENDIF
 READ
 STORE VAL(MNR:FACT) TO MNR:FACT
 @ 9,15 SAY MNR:FACT
 @ 9,63 SAY MDATA:FACT
 IF !(OPER) = 'R' .OR. !(OPER) = '▲'
    @ 10,35 SAY MREFSTOR
 ENDIF
 IF !(OPER) = 'R'
    @ 11,13 SAY MMOTIVREF
 ENDIF
 STORE 0 TO FLAG
 IF MNR:FACT = 0
    ? ' NUMAR'FACTURA = 0 ? '
    STORE 1 TO FLAG
 ENDIF
 IF VAL(*(MDATA:FACT,1,2)) < 1 .OR. VAL(*(MDATA:FACT,1,2)) > 31;
      .OR. VAL(*(MDATA:FACT,3,2)) < 1 .OR. VAL(*(MDATA:FACT,3,2)) > 12;
      .OR. VAL(*(MDATA:FACT,5,2)) < 83
    STORE 1 TO FLAG
    ? ' DATA VALOARE ERONATA '
 ENDIF
 ACCEPT 'D PTR.CORECTII,(CR) PTR. CONTINUARE ' TO VCOR1
 @ 12,0 SAY CHR(27)+CHR(87)
ENDDO
IF FLAG = 1
   ? ' DATE ERONATE ????'
   STORE 0 TO CONTOR
   DO WHILE CONTOR < 70
    STORE CONTOR + 1 TO CONTOR
   ENDDO
```

Fig. 18.17, II.

```
 LOOP
ENDIF
STORE 12 TO LINIE
@ LINIE,0 SAY CHR(27)+CHR(87)
STORE STR(MNR:FACT,5) TO KEY
SELE SECO
FIND '&KEY'
IF # = 0
   ? CHR(20)+CHR(66)+'FACTURA INEXISTENTA IN BAZA DE DATE ';
    +CHR(20)+CHR(64)+CHR(07)+CHR(07)
   STORE 'N' TO VCOR1
   STORE 0 TO CONTOR
   DO WHILE CONTOR < 80
    STORE CONTOR + 1 TO CONTOR
   ENDDO
   LOOP
ENDIF
STORE '            ' TO DENOP
IF !(OPER) = 'A'
   STORE 'ANULARE ' TO DENOP
 ELSE
   IF !(OPER) = 'R'
      STORE 'REFUZ  ' TO DENOP
    ELSE
      IF !(OPER) = 'S'
         STORE 'STORNO ' TO DENOP
      ENDIF
   ENDIF
ENDIF
IF !(IMPRIM) = 'D'
   SET PRINT ON
   SET CONS OFF
   ? ' ==================================================================='
   ? ' NR FACTURA : ' + STR(NR:FACT,5) +' DATA OP ';
      +STR(DATA:FACT,6)
   ? ' -------------------------------------------------------------'
   SET PRINT OFF
   SET CONS ON
ENDIF
STORE 12 TO LINIE
@ 12,0 SAY CHR(27)+CHR(87)
IF !(MREFSTOR) = '1'
   IF !(OPER) = 'R' .OR. !(OPER) = 'S'
      LOCATE FOR NR:FACT = MNR:FACT
      DO WHILE .NOT. EOF
       IF VAL:ER # 0
          SELE PRIM
          APPEND BLANK
          REPL NR:FACT WITH S.NR:FACT, DEN:ELEM WITH S.DEN:ELEM,;
               DATA:FACT WITH VAL(MDATA:FACT), CANT:ER WITH S.CANT:ER,;
               VAL:ER WITH S.VAL:ER
          IF !(IMPRIM) = 'D'
             SET CONS OFF
             SET PRINT ON
             ? 'ELEM ' + DEN:ELEM +' CANITITATE ';
                +STR(CANT:ER,5) + ' VALOARE ' + STR(VAL:ER,12,2)
             SET CONS ON
             SET PRINT OFF
          ENDIF
          IF !(OPER) = 'R'
             REPL TIP:OP WITH 2, MOTIV:R WITH MMOTIVREF
```

Fig. 18.17, III.

```
            SELE SECO
            REPL MOTIV:R WITH MMOTIVREF
            REPL CANT:ER WITH 0, VAL:ER WITH 0.00
          ELSE
            REPL TIP:OP WITH 3
            SELE SECO
            REPL CANT:ER WITH 0, VAL:ER WITH 0.00
            REPL MOTIV:R WITH MMOTIVREF
          ENDIF
          SELE PRIM
          STORE LINIE + 1 TO LINIE
          IF LINIE > 23
             STORE 12 TO LINIE
             @ LINIE,0 SAY CHR(27)+CHR(87)
          ENDIF
          ? 'ELEMENT ' +DEN:ELEM + 'CANTITATE ' + '&DENOP';
             + STR(CANT:ER,5)+' VALOARE ' + STR(VAL:ER,12,2)
          STORE VAL:ER + VALFACT TO VALFACT
        ENDIF
        SELE SECO
        CONTINUE
       ENDDO
       STORE LINIE + 1 TO LINIE
       IF LINIE > 23
          STORE 12 TO LINIE
          @ LINIE,0 SAY CHR(27)+CHR(87)
       ENDIF
       IF !(IMPRIM) = 'D'
          SET PRINT ON
       ENDIF
       ? ' VALOARE FACTURA '+'&DENOP' + STR(VALFACT,12,2)
       IF !(IMPRIM) = 'D'
          SET PRINT OFF
       ENDIF
       STORE 0 TO CONTOR
       DO WHILE CONTOR < 70
        STORE CONTOR + 1 TO CONTOR
       ENDDO
     ENDIF
  ENDIF
  IF !(MREFSTOR) = 'P' .OR. !(OPER) = 'A'
     STORE 'D' TO SFELEM
     DO WHILE !(SFELEM) = 'D'
      STORE 'D' TO CORECT
      STORE 0.00 TO VALGRUP
      STORE 0 TO CANTGRUP
      STORE '        ' TO MDENELEM
      STORE LINIE + 1 TO LINIE
      IF LINIE > 23
         STORE 12 TO LINIE
         @ LINIE,0 SAY CHR(27)+CHR(87)
      ENDIF
      @ LINIE,0 SAY 'ELEMENT ' GET MDENELEM
      READ
      STORE ' ' TO CORECTEL
      @ LINIE,9 SAY MDENELEM
      ACCEPT 'D-CORECTII (CR) CONTINUARE ' TO CORECTEL
      IF !(CORECTEL) = 'D'
         @ LINIE,0 SAY CHR(27)+CHR(87)
         LOOP
      ENDIF
      @ LINIE,0 SAY CHR(27)+CHR(87)
```

Fig. 18.17, IV.

```
      STORE 'D' TO VCOR4
      DO WHILE !(VCOR4) = 'D'
       @ LINIE,0 CHR(27)+CHR(87)
       STORE 'D' TO VCOR3
       DO WHILE !(VCOR3) = 'D'
        STORE '        ' TO MCANTGR
        STORE 0.00 TO MVALGR
        @ LINIE,0 SAY 'ELEMENT '+MDENELEM+'&DENOP'
        @ LINIE,30 SAY 'CANTITATE ' GET MCANTGR PICTURE '99999'
        @ LINIE,50 SAY 'VALOARE ' GET MVALGR PICTURE '999999999999'
        READ
        STORE VAL(MCANTGR) TO MCANTGR
        @ LINIE,40 SAY MCANTGR
        @ LINIE,61 SAY MVALGR
        IF MCANTGR # 0 .AND. MVALGR = 0
           ? ' VALOARE = 0 ? '
           STORE 0 TO CONTOR
           DO WHILE CONTOR < 70
            STORE CONTOR + 1 TO CONTOR
           ENDDO
           LOOP
        ENDIF
        STORE LINIE + 1 TO LINIE
        IF LINIE > 23
           STORE 12 TO LINIE
           @ LINIE,0 SAY CHR(27)+CHR(87)
        ENDIF
        ACCEPT 'D-CORECTII,(CR) CONTINUARE ' TO VCOR3
        @ LINIE,0 SAY CHR(27)+CHR(89)
        STORE LINIE - 1 TO LINIE
       ENDDO
       IF MCANTGR = 0 .AND. MVALGR = 0
          STORE 0 TO FLAG
          STORE 'N' TO VCOR4
          IF VALGRUP = 0
             STORE LINIE + 1 TO LINIE
             IF LINIE > 23
                STORE 12 TO LINIE
                @ LINIE,0 CHR(27)+CHR(87)
             ENDIF
             ? ' VALOARE = 0  LA NIVEL ELEMENT ?'
             STORE 0 TO CONTOR
             DO WHILE CONTOR < 70
              STORE CONTOR + 1 TO CONTOR
             ENDDO
             @ LINIE,0 CHR(27)+CHR(87)
             STORE LINIE - 1 TO LINIE
             STORE 1 TO FLAG
          ELSE
             STORE STR(NNR:FACT,5)+DEN:ELEM TO KEY
             SELE SECO
             FIND '&KEY'
             IF # = 0
                STORE LINIE + 1 TO LINIE
                IF LINIE > 23
                   STORE 12 TO LINIE
                   @ 12,0 SAY CHR(27)+CHR(87)
                ENDIF
                ? CHR(20)+CHR(66)+'ELEMENT INEXISTENT IN BAZA DE DATE';
                   +CHR(20)+CHR(64)+CHR(07)+CHR(07)
                STORE 0 TO CONTOR
                DO WHILE CONTOR < 70
                 STORE CONTOR + 1 TO CONTOR
                ENDDO
```

Fig. 18.17, V.

```
      @ LINIE,0 SAY CHR(27)+CHR(89)
      STORE 1 TO FLAG
      STORE LINIE - 1 TO LINIE
    ENDIF
  ENDIF
  IF FLAG = 0
    IF VAL:ER < VALGRUP
      STORE LINIE + 1 TO LINIE
      IF LINIE > 23
        STORE 12 TO LINIE
        @ LINIE,0 CHR(27)+CHR(87)
      ENDIF
      ? 'VALOARE IN FACTURA '+STR(VAL:ER,12,2)+'(E'+
       + 'VALOARE TASTATA '+STR(VALGRUP,12,2)+'(LR) PTR. CONTINUARE'
      WAIT
      @ LINIE,0 SAY CHR(27)+CHR(89)
      STORE LINIE - 1 TO LINIE
      STORE 1 TO FLAG
    ENDIF
  ENDIF
  IF FLAG = 0
    STORE VALGRUP + VALFACT TO VALFACT
    SELE PRIM
    APPEND BLANK
    REPL VAL:ER WITH VALGRUP, CANT:ER WITH CANTGRUP,;
        NR:FACT WITH NNR:FACT, DATA:FACT WITH VAL(NDATA:FACT),;
        DEN:ELEM WITH NDENELEM
    IF !(OPER) = 'R'
      REPL MOTIV:R WITH NMOTIVREF
      REPL TIP:OP WITH 2
      SELE SECO
      REPL CANT:ER WITH (CANT:ER -CANTGRUP),;
          VAL:ER WITH (VAL:ER - VALGRUP)
          REPL MOTIV:R WITH NMOTIVREF
    ELSE
      IF !(OPER) = 'S'
        REPL TIP:OP WITH 3
        SELE SECO
        REPL CANT:ER WITH (CANT:ER -CANTGRUP),;
            VAL:ER WITH (VAL:ER - VALGRUP)
        IF .NOT. (MOTIV:R $ ' ' .AND. VAL:EI > VAL:ER)
          REPL MOTIV:R WITH ' '
        ENDIF
      ELSE
        IF !(OPER) = 'A'
          REPL TIP:OP WITH 4
          SELE SECO
          REPL CANT:ER WITH (CANT:ER +CANTGRUP),;
              VAL:ER WITH (VAL:ER + VALGRUP)
          IF .NOT. (MOTIV:R $ ' ' .AND. VAL:EI > VAL:ER)
            REPL MOTIV:R WITH ' '
          ENDIF
        ENDIF
      ENDIF
    ENDIF
    STORE LINIE + 1 TO LINIE
    IF LINIE > 23
      STORE 12 TO LINIE
      @ LINIE,0 SAY CHR(27)+CHR(87)
    ENDIF
    IF !(IMPRIM) = 'D'
      SET PRINT ON
    ENDIF
    ? ' TOTAL '+'&DENOP'+ 'ELEM '+NDENELEM + ' CANT '+;
      +STR(CANTGRUP,5) +'VALOARE '+STR(VALGRUP,12,2)
    IF !(IMPRIM) = 'D'
      SET PRINT OFF
    ENDIF
  ENDIF
```

Fig. 18,17, VI.

```
      STORE LINIE + 1 TO LINIE
      IF LINIE > 23
        STORE 12 TO LINIE
        @ LINIE,0 SAY CHR(27)+CHR(87)
      ENDIF
      ACCEPT 'MAI AVETI ELEMENTE ? D/N ' TO SFELEM
      @ LINIE,0 SAY CHR(27)+CHR(87)
      STORE LINIE - 1 TO LINIE
    ELSE
      STORE MCANTGR +CANTGRUP TO CANTGRUP
      STORE MVALGR + VALGRUP TO VALGRUP
      STORE LINIE + 1 TO LINIE
      IF LINIE > 23
        STORE 12 TO LINIE
        @ LINIE,0 SAY CHR(27)+CHR(87)
      ENDIF
    ENDIF
  ENDDO GRUP
  ENDDO SFELEM
  STORE LINIE + 1 TO LINIE
  IF LINIE > 23
    STORE 12 TO LINIE
    @ LINIE,0 SAY CHR(27)+CHR(87)
  ENDIF
  @ LINIE,0 SAY 'VALOARE '+'&DENOP'+' '+'FACTURA '+STR(VALFACT,12,2)
  IF !(IMPRIM) = 'D'
    SET CONS OFF
    SET PRINT ON
    ? ' TOTAL '+'&DENOP' + ' FACTURA '+STR(VALFACT,12,2)
    SET CONS ON
    SET PRINT OFF
  ENDIF
  STORE 0 TO CONTOR
  DO WHILE CONTOR < 70
    STORE CONTOR + 1 TO CONTOR
  ENDDO
ENDIF P
IF !(OPER) = 'R'
  IF !(MREFSTOR) = 'I'
    STORE TVFRI + VALFACT TO TVFRI
  ELSE
    IF !(MREFSTOR) = 'P'
      STORE TVFRP + VALFACT TO TVFRP
    ENDIF
  ENDIF
ELSE
  IF !(OPER) = 'S'
    IF !(MREFSTOR) = 'I'
      STORE TVSFI + VALFACT TO TVSFI
    ELSE
      IF !(MREFSTOR) = 'P'
        STORE TVSFP + VALFACT TO TVSFP
      ENDIF
    ENDIF
  ENDIF
ENDIF
IF !(OPER) = 'A'
  STORE VALFACT + TASRP TO TASRP
ENDIF
ENDDO SF
RETURN
```

Fig.18,17, VII.

Atît în procedura din figura 18.14 cît și în cea din figura 18.17 s-a considerat că pentru un element din factură pot exista unul sau mai multe poziții. Datele dintr-o factură se înregistrează în două fișiere după structura prezentată în figurile 18.13 *a* și 18.13 *b*, unul fiind secundar (SECONDARY) iar altul primar (PRIMARY).

Orice operație privind facturile cu elementele prezentate în structură din figura 18.16 se înregistrează în ceea ce s-a numit fișierul de istoric al facturilor emise, refuzate, stornate sau anulate.

Cele două proceduri constituie exemple concrete dintr-o versiune a unui contract de cercetare mai complex [133] care mai cuprinde proceduri de operare a operațiilor privind încasările și storno-încasări, consultările și editările, editările de rapoarte și operațiile de închidere de lună.

A n e x ă

Din categoria instrumentelor lingvistice un loc important îl ocupă limbajele folosite în codificare.

În cadrul acestei anexe vom face o analiză succintă a acestora pentru a pune în evidență faptul că ele reflectă arhitectura actualelor calculatoare tip von Newman și pentru a desprinde din aceasta necesitatea construirii și a altor categorii de instrumente lingvistice orientate spre ceea ce literatura de specialitate numește instrumente lingvistice ale programării funcționale.

Limbajele COBOL, FORTRAN, PASCAL, CLU, ADA, mai sînt numite și limbaje imperative. Unitatea de lucru într-un program scris în aceste limbaje este *enunțul*. Pentru a atinge rezultatele dorite cu ajutorul unui program trebuie combinate enunțurile individuale.

Arhitectura actualelor calculatoare a jucat și joacă un rol important în alcătuirea limbajelor imperative. Această influență se poate vedea dacă analizăm următoarele trei caracteristici predominante ale acestora (tratate pe larg în [106]).

1. *Variabilele*. O componentă principală a structurii calculatoarelor este memoria, care constă dintr-un număr foarte mare de *celule*. Memoria este locul în care se înmagazinează datele. Celulele de memorie trebuie să aibă nume pentru a se putea ști unde sînt memorate datele. În limbajele evoluate noțiunile de bază ca celulă de memorie și nume sînt reprezentate de conceptul de variabilă. Variabila într-un limbaj de programare este practic o celulă de memorie căreia i s-a dat un nume și în care s-au înregistrat valori.

Așa se face că dacă scopul programării este de a produce valori, nu ne referim numai la valori ci și la celulele în care se află valorile ce ne interesează.

2. *Operația de repartizare*. Fiecare valoare calculată trebuie repartizată unei celule. Aceasta explică întîietatea pe care trebuie să o aibă enunțul de repartizare în limbajele de programare. Noțiunile de *celulă de memorare* și repartizare, sau așa cum s-a încetățenit *atribuire*, predomină în toate limbajele de programare și impun programatorului un stil de gîndire modelat cu detaliile arhitecturii acestei categorii de calculatoare.

3. *Repetiția*. Un program în limbaj imperativ își îndeplinește sarcinile de obicei executînd în mod repetat un șir de etape elementare, ceea ce este esența arhitecturii von Newman în care instrucțiunile sînt stocate în memorie.

Trebuie precizat faptul că în realizarea de limbaje evoluate se caută să ascundă nivelul inferior al mașinii von Newman. Un exemplu elocvent este oferit de folosirea expresiilor care permit programatorilor să păstreze anonime rezultatele intermediare (ele ascund faptul că toate celulele de memorie trebuie numite).

Expresiile încep să joace un rol mai mare decît enunțurile. Spre exemplu, limbajul ALGOL 68 cere ca fiecare construcție să fie o expresie, adică să aibă o valoare care poate fi folosită în alte expresii.

Una dintre cele mai mari probleme ridicate de limbajele imperative se referă la corectitudinea programelor.

Această dificultate este cauzată de faptul că acurateţea unui program în general depinde de conţinutul fiecărei celule a memoriei în parte. Pentru a respecta etapele de calcul, aşa cum se succed ele în timp, trebuie luate „instantanee" ale memoriei la fiecare pas, după fiecare instrucţiune.

Cele de mai sus justifică necesitatea unui stil de programare bazat pe funcţiuni şi pe matematică, mai degrabă decît pe memoria lui von Newman, pe acţiuni elementare şi pe repetiţie. Desigur că o asemenea necesitate presupune construirea unor instrumente software adecvate.

Se conturează tot mai mult necesitatea definirii unor funcţii precum şi a combinării acestor funcţii pentru a obţine funcţii şi mai puternice.

Se spune despre un limbaj de programare că este funcţional (sau aplicativ) dacă face uz de proprietăţile matematice ale funcţiilor.

Un astfel de limbaj are ca principale componente: un grup de funcţii primitive; un grup de forme funcţionale; un grup de obiecte de date; operaţia de aplicare.

În afară de cele patru componente de mai sus, un limbaj funcţional furnizează şi mecanismul de atribuire a unui nume funcţiilor noi în curs de definire evitînd astfel repetarea definirii funcţiei de fiecare dată cînd se aplică funcţia.

John Backus cel care a gîndit FORTRAN-ul, concepe un limbaj foarte simplu pentru programarea funcţională.

Obiectele limbajului sînt foarte simple: un obiect poate fi un *atom* sau o *înşiruire* (de obiecte). Şirul gol este cunoscut ca *Nil*. Atomii sînt şiruri de caractere ca de exemplu: A, ABC, 26 etc. Există un grup de funcţii şi un grup de forme funcţionale.

După J. Backus [106] limbajul funcţional trebuie să furnizeze *funcţii primitive* puternice şi de o largă aplicabilitate. Cu cît un limbaj are mai multe forme funcţionale, cu atît este mai „expresiv", întrucît formele funcţionale sînt mecanismele de deservire a programelor.

Limbajele pur funcţionale sînt cercetate pentru a servi ca bază în vederea definirii unei noi generaţii de limbaje de programare.

Unele din limbajele în uz curent prezintă însemnate trăsături funcţionale. Dintre acestea se consideră că LISP şi PROLOG sînt cel mai apropiate de limbajele funcţionale. Facilităţi de genul celor oferite de limbajele funcţionale, pun la dispoziţia programatorilor şi limbajul APL.

În încheierea referirilor cu privire la cele două categorii de limbaje menţionăm că limbajele imperative sînt mai eficace din punctul de vedere al timpului de execuţie, pentru că ele reflectă structura şi operaţiile maşinii. Totuşi ele cer programatorului să-şi îndrepte atenţia asupra unor detalii mecanice.

Această influenţă se poate vedea în stilul de programare practicat. Stilul de programare funcţională poate face programarea uşoară.

Instrumentele de date care sînt simple (spre exemplu, şirurile, listele etc.) permit conceperea unor structuri de date, fără a purta grija celulelor de memorare, valorile sînt mai puţin atribuite şi mai mult produse prin aplicarea funcţiilor şi transmise mai departe altor funcţii iar formele şi operaţiile funcţionale care se atribuie asupra instrumentelor de date reduc dependenţa lor de repetiţie. Această uşurinţă la nivel programare se răsfrînge însă într-o eficienţă scăzută în execuţie.

Trebuie însă să precizăm că şi criteriile de eficacitate pot fi cu totul altele la un alt model de maşină. Se poate ridica următoarea întrebare — Cum este mai bine? Să construim limbajele de programare pe criterii de eficacitate

a unei anumite maşini (în sens de calculator) sau să construim maşina astfel încît să sprijine un anumit limbaj sau o anumită clasă de limbaje? Este cert că actualele tipuri de calculatoare sprijină îndeosebi limbajele imperative care, de altfel, sînt abstracţii clădite pe baza acestei arhitecturi de calculatoare.

Necesar ar fi să se conceapă un limbaj pe baza unor strategii pe care noi le cunoaştem ca fiind apte de a rezolva probleme. Apoi s-ar putea concepe o maşină (în sensul unui calculator) care să susţină şi dezvolte limbaje. Desigur că în contextul arhitecturii actuale, considerentele de eficacitate vor elimina limbajele pur funcţionale de genul celui conceput de J. Backus. La acestea se adaugă şi puţina experienţă acumulată cu privire la ceea ce se numeşte programarea funcţională şi limbajele care să o răspîndească.

Sînt încă numeroase probleme de mare importanţă de rezolvarea cărora depinde şi structura viitoarelor limbaje. Problema cum ar fi a prelucrării grafice; intrări şi ieşiri aşteaptă rezolvări din cele mai eficiente.

Nu trebuie uitat însă că succesul unui limbaj este determinat, într-o mare măsură, de posibilitatea de a fi întrebuinţat de un mare număr de programatori obişnuiţi şi în orice împrejurare.

Nu trebuie să omitem nici perspectivele deschise de ceea ce în literatura de specialitate numeşte *a cincea generaţie de calculatoare* (primul prototip este prevăzut să apară în 1990). *Limbajele de programare de nivel foarte înalt* prevăzute a se utiliza pe această categorie de calculatoare oferă metodologii de programare îmbunătăţite specificînd mai degrabă *ce* urmează a fi realizat decît în mod tradiţional *cum* urmează a fi realizat. Categoriile cele mai dezvoltate de limbaje la nivel foarte înalt sînt limbaje funcţionale, ca de exemplu LISP, şi sînt sprijinite de limbaje logice ca de exemplu PROLOG.

Limbajele funcţionale şi limbajele logice sînt adecvate sistemelor specializate de programe bazate pe cunoştinţe.

Postfață

În această carte, structurată pe două volume, în baza unei analize a stadiului elaborării software-ului în țara noastră și pe plan mondial și luînd în calcul cerințele creșterii eficienței în acest domeniu de activitate în plină expansiune sînt mai întîi prefigurate căile și direcțiile de acțiune, apoi descrise modalitățile și soluțiile pentru un nou cadru metodologic, tehnologic, organizatoric și de conducere în programare.

Tehnicile, instrumentele, procedeele, algoritmii, factorii umani, modelele de organizare și conducere sînt prezentate pe structura cadru a etapelor ciclului de viață al programelor.

Tendința spre sisteme de lucru în timp real, bazate pe interacțiunea om-calculator, mutațiile pe care acestea le produc, atît în elaborarea programelor cît și în utilizarea calculatoarelor*) au făcut ca ingineria programării să cuprindă elemente specifice tipului de software ce urmează să fie construit.

Se poate vorbi, spre exemplu, de ingineria programării pentru sisteme software cu lucru în timp real, ingineria programării pentru sisteme de gestiune a bazelor de date ș.a.m.d.

Cartea de față, fiind prima de acest gen în literatura de specialitate românească, s-a limitat doar la ceea ce am putea numi cadrul general al ingineriei programării, lăsînd pentru preocupări viitoare tratarea elementelor specifice ingineriei programării pentru sisteme software specializate.

Cu titlu informativ cităm unele cercetări efectuate pe această temă în cadrul ITCI I.P.B., ș.a., unele din ele făcute cunoscute cititorilor prin intermediul unor lucrări din care selectăm [93], [100], [102], [112], [127], [129], [130], [131].

Realizările obținute pe linia elaborării programelor cu asistență din partea calculatorului (în memoria căruia sînt înregistrate informații, raționamente și deducții) constituie, de asemenea, un factor cu implicații importante în ingineria programării.

A patra generație de calculatoare, specifică deceniului anilor '80, caracterizată prin tendința pronunțată spre interactivitate și prelucrare distribuită și a cincea generație anunțată în lume pentru deceniul anilor '90, care se bazează pe arhitecturi și structuri noi de calculatoare, ridică noi probleme ceea ce va impune și soluții noi în planul ingineriei programării. Trecerea la prelucrarea electronică a datelor la prelucrarea inteligentă a cunoștințelor produce mutații importante și în activitatea de programare.

Limbajele de programare de nivel foarte înalt prevăzute a se utiliza pe calculatoarele din generația a cincea oferă metodologii de programare îmbunătățite, specificînd mai degrabă *ce* urmează a fi realizat decît în mod tradițional *cum* urmează a fi realizat. Categoriile cele mai dezvoltate de limbaje la nivel foarte înalt sînt funcționale, ca de exemplu LISP, și sînt sprijinite de limbaje logice ca de exemplu PROLOG [106].

---

*) Se vorbește în primul rînd despre umanizarea interfeței om-calculator.

Sistemele de calculatoare din a cincea generație care sînt în studiu pot fi considerate ca formînd o nouă familie de calculatoare în care membrii oferă modalități puternice pentru rezolvări de probleme, deducții, prelucrări date de cunoștințe, intrări-ieșiri inteligente. Se estimează că aceste calculatoare vor permite conversații sub formă de vorbire, grafice, limbaj natural etc., posibilități care să ducă la schimbul de informații într-o formă specifică oamenilor.

Sistemele software specifice acestei categorii de calculatoare constau dintr-un grup de sisteme care vor permite proiectarea și producerea de sisteme optime de prelucrare a informațiilor pentru diverse aplicații vor avea cunoștințe în legătură cu ceea ce urmează a fi produs, cu sursele de producție etc. Sistemele software vor include subsisteme care conduc, de la o specificație scrisă la un produs final, subsisteme pentru verificarea corectitudinii și subsisteme pentru simularea operațiilor. În plus ele vor include trei subsisteme de sprijin și anume: un sistem inteligent de programare pentru manipularea programelor, un sistem de proiectare a bazei de cunonștințe pentru mînuirea unei baze de cunoștințe și un sistem de proiectare inteligent VLSI.

Sistemul software urmează, de asemenea, să includă funcții puternice pentru ușurarea folosirii sistemului însuși.

Progresele înregistrate și nivelul la care s-a ajuns în știința calculatoarelor prefigurează medii de programare moderne și-n consecință soluții în plan tehnologic, metodologic, instrumente organizatorice și de conducere capabile să ducă la o creștere considerabilă a productivității muncii în programare și la obținerea unor produse software performante la un cost redus.

BIBLIOTECA TEHNICĂ
UZINA MECANICĂ IASI